

SYSTEM, METHOD AND APPARATUS FOR SUPPORTING A KERNEL MODE DRIVER

Field of the Invention

5 The present invention generally relates to management instrumentation systems, and more specifically relates to computer systems having instrumented hardware devices.

Background of the Invention

Background of WBEM

10 Corporations and other enterprises have a need to monitor the performance and status of elements of their computer networks to prevent data loss and to maximize resource efficiency. The computer industry is addressing that need by putting together the concept of Web-Based Enterprise Management ("WBEM"). WBEM is an industry initiative to develop a standardized, nonproprietary means for accessing and sharing management information in an enterprise network. The WBEM initiative is intended to solve the problem of collecting end-to-end management and diagnostic data in enterprise networks that may include
15 hardware from multiple vendors, numerous protocols and operating systems, and a legion of distributed applications

The founding companies of the WBEM initiative developed a prototype set of environment-independent specifications for how to describe and access any type of management instrumentation, including existing standards such as Simple Network
20 Management Protocol and Desktop Management Interface. A core component of the specification is a standard data description mechanism known as the Common Information Model ("CIM"). The CIM specification describes the modeling language, naming, and

mapping techniques used to collect and transfer information from data providers and other management models. The Windows Management Instrumentation ("WMI") system is a Windows-based implementation of the CIM specification and is fully compliant with the WBEM initiative.

5 One component of WMI is the Extensions to the Windows Driver Model ("WDM") provider (the "WMI provider") for kernel component instrumentation. The WMI provider interfaces with a kernel mode driver, coded in accordance with the Extensions to WDM specification, to pass WMI data between user mode and kernel mode. WMI uses the WMI provider to publish information, configure device settings, and supply event notification from
10 device drivers.

Identification of the Problem

Although the WMI provider is a key component in making the WMI system work, it is not without disadvantages. First, manufacturers must add substantial additional code to their device drivers to support the WMI system. At present, each manufacturer must
15 independently develop software methods and functions to incorporate in their device drivers to support the WMI Extensions to WDM specification. This creates a burden shared by every developer of device drivers intended to be used with the WMI system. It takes additional time for each developer to produce both the code specific to the developer's device, and the code specific to the WMI system. Second, because similar code is used in
20 each device driver to support WMI, many instances of functionally-identical code are loaded in memory by the several drivers. The result is an inefficient operating state containing more system overhead than needed to support WMI. Overall system performance may suffer. Third, the likelihood of coding errors, or "bugs," is increased when many disparate vendors develop code to perform substantially the same function.

25 Accordingly, a need exists for a mechanism that allows disparate device drivers intended to interface with the WMI system to share code designed to operate with the WMI system.

Summary of the Invention

The present invention addresses the above described needs and disadvantages by
30 providing a set of common software routines that may be accessed by device drivers in support of the WMI system. The set of common routines includes typical routines that would ordinarily be executed by device drivers designed in accordance with WMI. The common routines may reside in a library, dynamically accessible by the device drivers. When a device driver receives a message from the WMI system, the device driver may pass the message to
35 the library to be handled in a common manner. In this manner, the developers of device

drivers in accordance with the WMI system need only develop so much code as is necessary to support any unique features or data storage of its associated hardware. The result is shortened development time and fewer programming errors. In addition, the overall system performance may be improved because fewer instances of similar code are loaded in memory to support the WMI system.

While the preferred implementation of the present invention provides a dynamically linked library, some driver standards, such as the Small Computer Systems Interface ("SCSI") miniport standard, do not allow for accessing code in a dynamically linked library. For those drivers, the library may be included as a static part of the driver at link-time. Although this solution may still result in multiple instances of the same code in memory, the development time is still shortened, and the typicality of the code results in a more stable WMI and Windows system. Also, the use of the library allows the underlying WMI infrastructure to be modified without affecting the developer's driver so long as the interface to the library is maintained.

Brief Description of the Drawings

The foregoing aspects and many of the attendant advantages of this invention will become more readily appreciated as the same becomes better understood by reference to the following detailed description, when taken in conjunction with the accompanying drawings, wherein:

FIGURE 1 is a functional block diagram of a computer suitable for providing an exemplary operating environment for the present invention;

FIGURE 2 is a functional block diagram of software components embodying the present invention resident on the computer system of FIGURE 1;

FIGURE 3 is a functional block diagram illustrating the concept of moving typical code from multiple drivers to a common library in accordance with the present invention;

FIGURE 4 is a functional block diagram illustrating the concept of a driver stack serviced by the common library of FIGURE 3;

FIGURE 5 is an event trace illustrating the flow of processing that occurs in a common library system in accordance with the present invention;

FIGURE 6 is a logical flow diagram illustrating steps performed by a process for utilizing a common driver library in accordance with the present invention; and

FIGURE 7 is a logical flow diagram illustrating steps performed by a process for generating an event message through the use of a common driver library, in accordance with the present invention.

Detailed Description of the Preferred Embodiment

The present invention is directed to a system and method for supporting a system of kernel mode device drivers that share common code by moving that common code to a software library. The present invention may be embodied in a management instrumentation system, such as the WMI system promoted by the Microsoft Corporation of Redmond, Washington.

Exemplary Operating Environment

FIGURE 1 and the following discussion are intended to provide a brief, general description of a suitable computing environment in which the invention may be implemented.

While the invention will be described in the general context of an application program that runs on an operating system in conjunction with a personal computer, those skilled in the art will recognize that the invention also may be implemented in combination with other program modules. Generally, program modules include routines, programs, components, data structures, etc. that perform particular tasks or implement particular abstract data types. Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including hand-held devices, multiprocessor systems, microprocessor-based or programmable consumer electronics, minicomputers, mainframe computers, and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

Referring to FIGURE 1, an exemplary system for implementing the invention includes a conventional personal computer 20, including a processing unit 21, a system memory 22, and a system bus 23 that couples the system memory to the processing unit 21. The system memory 22 includes read only memory (ROM) 24 and random access memory (RAM) 25. A basic input/output system 26 (BIOS), containing the basic routines that help to transfer information between elements within the personal computer 20, such as during start-up, is stored in ROM 24. The BIOS 26 may additionally store AML code for use in conjunction with an associated ACPI device. The personal computer 20 further includes a hard disk drive 27, e.g. to read from or write to a hard disk 39, a magnetic disk drive 28, e.g., to read from or write to a removable disk 29, and an optical disk drive 30, e.g., for reading a CD-ROM disk 31 or to read from or write to other optical media. The hard disk drive 27, magnetic disk drive 28, and optical disk drive 30 are connected to the system bus 23 by a hard disk drive interface 32, a magnetic disk drive interface 33, and an optical drive interface 34, respectively. The drives and their associated computer-readable media provide

nonvolatile storage for the personal computer 20. Although the description of computer-readable media above refers to a hard disk, a removable magnetic disk and a CD-ROM disk, it should be appreciated by those skilled in the art that other types of media which are readable by a computer, such as magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, and the like, may also be used in the exemplary operating environment.

A number of program modules may be stored in the drives and RAM 25, including an operating system 35, one or more application programs 36, a driver library 37 constructed in accordance with one embodiment of the present invention, and program data 38. A user may enter commands and information into the personal computer 20 through a keyboard 40 and pointing device, such as a mouse 42. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 21 through a serial port interface 46 that is coupled to the system bus, but may be connected by other interfaces, such as a game port or a universal serial bus (USB). A monitor 47 or other type of display device is also connected to the system bus 23 via an interface, such as a video adapter 48. In addition to the monitor, personal computers typically include other peripheral output devices (not shown), such as speakers or printers.

The personal computer 20 may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer 49. The remote computer 49 may be a server, a router, a peer device or other common network node, and typically includes many or all of the elements described relative to the personal computer 20, although only a memory storage device 50 has been illustrated in Fig. 1. The logical connections depicted in Fig. 1 include a local area network (LAN) 51 and a wide area network (WAN) 52. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet.

When used in a LAN networking environment, the personal computer 20 is connected to the LAN 51 through a network interface 53. When used in a WAN networking environment, the personal computer 20 typically includes a modem 54 or other means for establishing communications over the WAN 52, such as the Internet. The modem 54, which may be internal or external, is connected to the system bus 23 via the serial port interface 46. In a networked environment, program modules depicted relative to the personal computer 20, or portions thereof, may be stored in the remote memory storage device. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

FIGURE 2 is a functional block diagram of software components embodying the present invention resident on the computer 20 of FIGURE 1. Illustrated is a management system 200, including multiple management applications 201 executing in user mode 203. The management system 200 may be any CIM schema compliant management system, such as the WMI management system described above. Although embodiments of the present invention may be described here in cooperation with the WMI management system, the present invention is equally applicable to other management systems. Reference here to the WMI management system is for illustrative purposes only, and does not limit the applicability of the invention.

Interfacing with the management applications 201 is a WMI agent 207. The WMI agent 207 maintains and provides access to a WMI store 209, which is a database containing the management information exposed by the management system 200. The management information contained in the WMI store 209 comes from multiple providers, such as components 211, 212, and 213. The providers act as intermediaries between the WMI agent 207 and one or more managed objects. When the WMI agent 207 receives a request from a management application 201 for information that is not available from the WMI store 209, or for notification of events that are unsupported, the WMI agent 207 forwards the request to an appropriate provider. That provider then supplies the information or event notification requested.

One such provider is the WMI Extensions to Windows Driver Model ("XWDM") provider (the "WMI provider") 214. The WMI provider 214 includes two parts: a user mode component ("UM component") 215 and a kernel mode component ("KM component") 217. The UM component 215 communicates with the KM component 217 to pass messages between the user mode 203 and the kernel mode 219. The WMI provider 214 allows instrumented devices to make management information available to the management system 200, and hence management applications 201, by providing a pipeline between the user mode 203 and the kernel mode 219.

In kernel mode 219, several device drivers, such as driver 221 and driver 222, support their associated devices, such as device 223 and device 224, respectively, and pass information to the management system 200 via the WMI provider 214. The drivers operate in conjunction with the management system 200 to allow the management applications to query or set management information within the several instrumented devices. In addition to queries and sets, the management system allows WMI method calls, which are functionally equivalent to an I/O control ("IOCTL") call to a device.

The WMI provider 214 and the device drivers 221, 222 communicate by passing I/O Request Packets ("IRP") 227. The IRPs 227 are instructions to perform actions related to the operation of the management system 200. For instance, a particular IRP 227 may instruct the driver 221 to begin collecting data on its associated device 223. Another IRP 227 may instruct the driver 221 to end collecting that data. Several of the IRPs used by the WMI management system are detailed in the attached appendix, and are incorporated herein by reference for illustrative purposes only.

Also illustrated is a driver library 37 constructed in accordance with the present invention. The driver library 37, named "WMILIB" in this example, is a kernel mode software library that includes software routines that would ordinarily be included in each of multiple device drivers, such as both in driver 221 and driver 222. The kernel mode device drivers, such as driver 221, may call the driver library 37 to request that many routine functions be performed by the driver library 37 rather than by the individual device drivers. The driver library 37 may also call back to the kernel mode drivers and request certain device-specific information, performance or request a specific action. The interaction of the WMI provider 214, the kernel mode device drivers, and the driver library 37 is illustrated in FIGURE 3 and described in detail below.

FIGURE 3 is a functional block diagram illustrating in greater detail the interaction between the WMI provider 214, the kernel mode device drivers, and the driver library 37 to achieve the benefits of the present invention. To begin, the WMI provider 214 issues an IRP to a kernel mode device driver, such as IRP 301 to driver 221. IRP 301 may be an instruction to set data within the device 223 associated with the driver 221, it may be an instruction to retrieve data, or it may be an instruction for the driver 221 to cause the device 223 to perform some function. The code that would ordinarily handle the IRP 301 is typical code 302 that also resides in each of several other kernel mode device drivers, such as driver 222. However, in accordance with this embodiment of the invention, the typical code 302 actually resides in the driver library 37 rather than in the separate kernel mode device drivers. For that reason, rather than handle the IRP 301 directly, the driver 221 passes the IRP 301 to the driver library 37. The driver library 37 of this embodiment is accessible to the other drivers by way of several Application Programming Interface ("API") calls. Exemplary API calls used in connection with the WMI management system are described in detail in the attached appendix, and are incorporated herein by reference for illustrative purposes only.

In this manner, the driver library 37 may perform many functions that otherwise would be performed by the several kernel mode device drivers. However, the device drivers may

also require some unique code, such as the unique code 307 associated with the driver 221 or the unique code 309 associated with the driver 222. It should be noted that unique code 307 is different from unique code 309. For example, unique code 307 may provide access to data registers or other features associated with the device 223, but which are inapplicable to another device, such as device 224. Consequently, each device driver maintains that software code necessary for interfacing to its associated device.

To handle the IRP 301, the driver library 37 may require access to the unique code 307, 309 maintained by the device drivers. For example, to handle the IRP 301, the driver library 37 may require access to data stored in a register on the device 223 itself. In that case, the driver library 37 may call back to the driver 221 to execute the unique code 307 and retrieve the requested data or perform an action. Exemplary callback routines used in connection with the WMI management system are described in detail in the attached appendix, and are incorporated herein by reference for illustrative purposes only.

FIGURE 4 is a functional block diagram illustrating an alternative embodiment of the present invention as it may be applied to a driver 222 that contains multiple drivers. In this embodiment, driver 222 is actually a driver stack, and includes more than one driver acting in concert to support the same device 224. One example of a driver stack may be a driver intended to interface with a SCSI device. Such a driver may employ both a SCSI port driver 401 and a SCSI miniport driver 403. The SCSI miniport driver 403 is a special kind of device driver designed to work in conjunction with the SCSI port driver 401 to support a SCSI device, such as device 224. The SCSI port driver 401 supplies the interface to the operating system 35 and some common code, while the SCSI miniport driver 403 contains any hardware specific code.

As is known to those skilled in the art, the SCSI miniport driver 403 cannot call code other than the SCSI port driver 401, and, for that reason, is unable to access the driver library 37 dynamically. Moreover, if the SCSI miniport driver 403 were modified to call the SCSI port driver 401 for functions similar to those provided by the driver library 37, then the SCSI miniport driver 403 would be unable to interface with earlier versions of the SCSI port driver 401. For those reasons, this embodiment of the invention provides a static driver library 37', rather than a dynamic library, that is incorporated into the SCSI miniport driver 403 at link time. The code from the driver library 37 is included in the SCSI miniport driver 403 as a static driver library 37', and the SCSI miniport driver 403 may directly access any necessary routines from the static driver library 37'.

As depicted in FIGURE 4, the management system 200 issues to the driver 222 an IRP 411. The SCSI port driver 401 receives the IRP 411 and first determines whether the

IRP 411 is intended for it. If the SCSI port driver 401 is intended to handle the IRP 411, then it does so. If not, then the SCSI port driver 401 translates the IRP 411 to a SCSI Request Block ("SRB") 413, which is a message format used with SCSI drivers, and passes the SRB 413 to the SCSI miniport driver 403. If the SRB 413 includes instructions that involve executing code related to the management system 200, the SCSI miniport driver 403 may call the static driver library 37' incorporated in the SCSI miniport driver 403. That configuration allows the SCSI miniport driver 403 to take advantage of the driver library 37 even though the SCSI miniport driver 403 cannot dynamically link to the driver library 37.

FIGURE 5 is an event trace illustrating the management system 200 supporting a driver constructed in accordance with the present invention. The event trace begins at step 501 when the management system 200 issues an IRP to the driver 221. The first IRP may be a simple request for data or other action that the driver can handle directly. For example, the first IRP may be a simple request for data which the driver can handle directly. For instance, the driver may be a filter driver configured to intercept IRPs intended for another driver, and which handles those intercepted IRPs directly. The code in the driver 221 may not need assistance to handle that IRP, and consequently, at step 502, the driver 221 handles the IRP directly and performs the requested action. The driver 221 may also return any requested data to the management system 200.

At step 503, the management system 200 may issue a second IRP to the driver 221. Unlike the first IRP, the second IRP may require additional input beyond the scope of the code within the driver 221. In that case, at step 504, the driver 221 passes a message to the driver library 37 identifying the particular IRP. In this case, it is possible that the driver library 37 can handle the second IRP without further intervention by the driver 221, and consequently, at step 505, the driver library 37 performs the action requested by the IRP on behalf of the driver and without further assistance of the driver. For example, the driver library 37 may return any data requested by the management system 200. Alternatively, the return may be simply an indication that the IRP has been handled.

At step 506, the management system 200 issues a third IRP to the driver 221. As with the second IRP, the driver 221 does not handle the particular IRP. Accordingly, as with the second IRP, the driver 221 passes the IRP to the driver library 37. However, unlike the second IRP, to handle the third IRP, the driver library 37 requires some subaction from the driver 221. For example, the IRP may request data stored within the device 223 and which must be retrieved using unique code 307 within the driver 221. Accordingly, at step 508, the driver library 37 may issue a callback to the driver 221 requesting that it perform some subaction, such as retrieving the data stored on the device 223. At step 509, the driver 221

performs the requested subaction. For instance, the driver 221 may execute the unique code 307 to retrieve the requested data and return, at step 509, that data to the driver library 37. The driver library 37 may then format that data in a way that the management system 200 expects, and finally complete the requested action at step 510. In this example, completing the requested action may involve returning the retrieved data to the management system 200.

FIGURE 6 is a logical flow diagram illustrating a process performed by one embodiment of the present invention to make use of the driver library 37 described above. The process begins at starting block 601, where the management system 200 begins to generate an instruction for a device driver, such as the driver 223. Processing continues at block 602.

At block 602, the management system 200 issues an IRP to the first driver in a driver stack. As mentioned above, a single device, such as device 223, may be serviced by a driver stack containing more than one device driver working in conjunction (called a "driver stack"). When an IRP is directed at information associated with a particular device, the IRP may actually be intended for a particular device driver in a driver stack, and should identify for which device driver in the stack the IRP is intended. Consequently, the management system 200 issues the IRP to the highest level driver (identified here as the first driver) in the driver stack, and processing continues at decision block 604.

At decision block 604, the current driver identifies whether the IRP is intended for that driver. The current driver may make that determination by comparing an identifier stored in the IRP to an identifier associated with the driver. If the IRP is not intended for the current driver, processing proceeds to block 606 where the IRP is passed to the next driver in the stack and decision block 604 is repeated. It should be noted that there may be only a single driver in the stack, in which case the IRP should be intended for that driver. If the IRP is intended for the current driver, processing proceeds to block 608.

At block 608, after the intended driver has been determined, that driver may pass the IRP to the driver library 37. As discussed above, it is not necessary to the proper operation of the present invention that a driver pass all IRPs to the driver library 37. As discussed above, developers of device drivers may choose to include code in the driver to handle particular IRPs, while calling the driver library 37 for others. Therefore, at block 608, it is envisioned that any IRPs not chosen to be handled directly by the driver be passed to the driver library 37. Processing then proceeds to decision block 610.

At decision block 610, the driver library 37 identifies whether the particular IRP requires data from or further action by the calling driver. For example, if the IRP is a request for particular data only available through the driver, the driver library 37 may decide to call

back to the driver for that information. At block 612, if any such information or input is required, the driver library 37 calls the driver for that information, and at decision block 610, the driver library 37 again determines whether further information is required. After receiving from the driver any additional information required to service the IRP, processing proceeds to
5 block 614.

At block 614, the driver library 37 executes the routines necessary to service the particular IRP. Many varying routines and functions may be performed to handle the particular IRP. For example, an IRP may be issued requesting that data values be changed. However, if the driver does not support changing data values then the driver library 37 may
10 return an error without the involvement of the driver. Another IRP may be issued requesting the driver library 37 to return all data associated with a driver, or a single instance of data associated with a particular device, such as device 223. As mentioned above, servicing the IRP may require actions from the driver in the form of data queries or sets related to the device. Likewise, the IRP may be a request to execute a method associated with data
15 exposed by the driver. These examples are provided to illustrate the nature of the functionality of the driver library 37, and those skilled in the art will appreciate that many other functions and routines may be provided within the driver library 37. When the IRP has been handled, processing terminates at block 616.

FIGURE 7 is a logical flow diagram illustrating steps performed by a process for
20 generating an event message through the use of driver library 37. The process begins at starting step 701, where an instrumented device 223 generates a notification that an event has occurred at the device 223. For example, if the device 223 is a temperature sensor, the event may be that the temperature of the computer 20 exceeds a given threshold. The process then continues at block 703.

25 At block 703, the device 223 issues a notification of the occurrence of the event to the driver 221. The notification of the event may take the form of an interrupt or other acceptable notification mechanism. Processing proceeds to block 705.

At block 705, the driver 221 passes to the driver library 37 the notification of the event with a request to handle that notification. For example, handling the event may include
30 generating a properly-formatted message for issuance to the management system 200. In addition, handling the event may include retrieving from the device 223 certain data associated with the event. Accordingly, to simply the burden on the driver 221 of handling the event, common functions for data formatting and message generation may be stored within the driver library 37 and called to assist in handling the event. Processing continues at
35 block 707.

At block 707, the driver library 37 may optionally call back to the driver 221 to retrieve any data associated with the event, such as a temperature value from a register within the device 223. The unique code 307 within the driver 221 may be invoked to retrieve and pass that data to the driver library 37. Any function provided by the unique code 307 may be invoked by the driver library 37. Processing continues at block 709.

At block 709, the driver library 37 may format any retrieved data in a buffer to be passed to the management system 200 along with an event notification message. For example, the management system 200 may expect data to be in a common format when passed with an event notification method. Code for constructing that common format may reside within the driver library 37, and therefore the data passed from the driver 221 may be raw, unformatted data. Processing continues at block 711.

At block 711, the driver library 37 issues to the management system 200 the event message constructed at block 709. Processing then terminates at ending block 713.

While the preferred embodiment of the invention has been illustrated and described, it will be appreciated that various changes can be made therein without departing from the spirit and scope of the invention.

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

Chapter 1 WMI IRPs

[This is preliminary documentation and subject to change.]

This chapter describes the Windows Management Instrumentation IRPs that are part of the WMI extensions to WDM. All WMI IRPs use the major code IRP_MJ_SYSTEM_CONTROL and a minor code that indicates the specific WMI request. The WMI kernel-mode component can send WMI IRPs any time following a driver's successful registration as a supplier of WMI data. WMI IRPs typically get sent when a user-mode data consumer has requested WMI data.

All drivers must set a dispatch table entry point that can be used by a DispatchSystemControl routine to handle WMI requests. If a driver registers as a WMI data provider by calling

IoWMIRegistrationControl, it must handle such requests in one of the following ways:

- Call the kernel-mode WMI library routines declared in the *wmilib.h* header file. Drivers can use these routines only if they base static instance names on a single base name string or the device instance ID of a PDO. Drivers that use dynamic instance names can not use the WMI library routines.
- Process and complete any request that was tagged with a pointer to the driver's device object. Such a request is passed by the driver in its call to **IoWMIRegistrationControl**. Other IRP_MJ_SYSTEM_CONTROL requests must be forwarded to the next-lower driver.

The WMI library routines simplify the handling of WMI requests. Instead of processing each WMI request, a driver calls **WmiSystemControl** with a pointer to its device object, the IRP, and a **WMI_LIB_CONTEXT** structure. This **WMI_LIB_CONTEXT** structure contains pointers to a set of **DpWmiXxx** callback routines that are defined by the driver. The WMI library validates the IRP parameters and calls the driver-provided **DpWmiXxx** routine for driver-specific processing. WMI library then packages any output in an appropriate **WNODE_XXX** structure. The output and status are returned to the caller. Drivers that use dynamic instance names must handle WMI requests by filling in the **WNODE_XXX** structure directly.

Drivers that do not register as WMI data providers must forward all WMI requests to the next-lower driver.

For information about registering as a WMI data provider, handling WMI IRPs, and using the WMI kernel-mode library routines, see the *Kernel-Mode Drivers Design Guide*.

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

IRP_MN_CHANGE_SINGLE_INSTANCE

[This is preliminary documentation and subject to change.]

file://C:\Documents and Settings\Administrator\ALANWAR-SLOWRONL... \-hh-D2D.lit 6/28/1999

All drivers that support WMI must handle this IRP.

When Sent

WMI sends this IRP to change all data items in a single instance of a data block.

WMI sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

Parameters.WMI.ProviderId points to the device object of the driver that should respond to the request. This pointer is found in the driver's I/O stack location in the IRP.

Parameters.WMI.DataPath points to a GUID that identifies the data block associated with the instance to be changed.

Parameters.WMI.BufferSize indicates the size of the nonpaged buffer at **Parameters.WMI.Buffer**.

Parameters.WMI.Buffer points to a **WNODE_SINGLE_INSTANCE** structure that identifies the instance and specifies new data values.

Output

None.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irmp->IoStatus.Status** and **Irmp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irmp->IoStatus.Status** to **STATUS_SUCCESS** or to an appropriate error status such as the following:

STATUS_WMI_INSTANCE_NOT_FOUND

STATUS_WMI_GUID_NOT_FOUND

STATUS_WMI_READ_ONLY

STATUS_WMI_SET_FAILURE

On success, the driver sets **Irmp->IoStatus.Information** to zero.

Operation

A driver that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its **WMI_LIB_CONTEXT** structure, a pointer to its device object, and a pointer to the

file://C:\Documents and Settings\Administrator\ALANWAR-SLOWRONL... \-hh-D2D.lit 6/28/1999

IRP. **WmiSystemControl** calls the driver's **DpWmiSetDataBlock** routine, or returns **STATUS_WMI_READ_ONLY** to the caller if the driver does not define an entry point for such a routine.

A driver that handles an **IRP_MN_CHANGE_SINGLE_INSTANCE** request does so only if the device object pointer at **Parameters.WmiProviderId** matches the pointer passed by the driver in its call to **IoWmiRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

If the driver handles the request, it first checks the GUID at **Parameters.WmiDataPath** to determine whether it identifies a data block supported by the driver. If not, the driver fails the IRP and returns **STATUS_WMI_GUID_NOT_FOUND**.

If the driver supports the data block, it checks the input **WNODE_SINGLE_INSTANCE** at **Parameters.WmiBuffer** for the instance name, as follows:

- If **WNODE_FLAG_STATIC_INSTANCE_NAMES** is set in **WnodeHeader.Flags**, the driver uses **InstanceIndex** as an index into the driver's list of static instance names for that block. WMI obtains the index from registration data provided by the driver when it registered the block.
- If **WNODE_FLAG_STATIC_INSTANCE_NAMES** is clear in **WnodeHeader.Flags**, the driver uses the offset at **OffsetInstanceName** to locate the instance name string in the input **WNODE_SINGLE_INSTANCE**. **OffsetInstanceName** is the offset in bytes from the beginning of the structure to a USHORT which is the length of the instance name string in bytes (not characters), including the NUL terminator if present, followed by the instance name string in Unicode.

If the driver cannot locate the specified instance, it must fail the IRP and return a **STATUS_WMI_INSTANCE_NOT_FOUND**. In the case of an instance that has a dynamic instance name, this status indicates that the driver does not "own" the instance. WMI can therefore continue to query other data providers and return an appropriate error to the data consumer if another provider finds the instance but cannot handle the request for some other reason.

If the driver locates the instance and can handle the request, it sets the read/write data items in the instance to the values in the **WNODE_SINGLE_INSTANCE**, leaving any read-only items unchanged. If the entire data block is read-only, the driver should fail the IRP and return **STATUS_WMI_READ_ONLY**.

If the instance is valid but the driver cannot handle the request, it can return any appropriate error status.

See Also

DpWmiSetDataBlock, **IoWmiRegistrationControl**, **WMI_LIB_CONTEXT**, **WmiSystemControl**, **WNODE_SINGLE_INSTANCE**

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

IRP_MN_CHANGE_SINGLE_ITEM

[This is preliminary documentation and subject to change.]

All drivers that support WMI must handle this IRP.

When Sent

WMI sends this IRP to change a single data item in a single instance of a data block.

WMI sends this IRP at **IRQL_PASSIVE_LEVEL** in an arbitrary thread context.

Input

Parameters.WmiProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters.WmiDataPath points to a GUID that identifies the data block to be set.

Parameters.WmiBufferSize indicates the size of the nonpaged buffer at **Parameters.WmiBuffer**.

Parameters.WmiBuffer points to a **WNODE_SINGLE_ITEM** structure that identifies the instance of the data block, the ID of the item to set, and a new data value.

Output

None.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to **STATUS_SUCCESS** or to an appropriate error status such as the following:

STATUS_WMI_INSTANCE_NOT_FOUND

STATUS_WMI_INSTANCE_ID_NOT_FOUND

STATUS_WMI_GUID_NOT_FOUND

STATUS_WMI_READ_ONLY

STATUS_WMI_SET_FAILURE

On success, a driver sets **Irp->IoStatus.Information** to zero.

Operation

A driver that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its **WMI_LIB_CONTEXT** structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's **DpWmiSetDataItem** routine, or returns **STATUS_WMI_READ_ONLY** to the caller if the driver does not define an entry point for such a routine.

A driver should handle an **IRP_MN_CHANGE_SINGLE_ITEM** request only if **Parameters.WmiProviderId** points to the same device object as the pointer that the driver passed to **IoWmiRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

Before handling a request, the driver determines whether **Parameters.WmiDataPath** points to a GUID that the driver supports. If it does not, the driver fails the IRP and returns **STATUS_WMI_GUID_NOT_FOUND**.

If the driver supports the data block, it checks the input **WNODE_SINGLE_ITEM** structure that **Parameters.WmiBuffer** points to for the instance name, as follows:

- If **WNODE_FLAG_STATIC_INSTANCE_NAMES** is set in **WnodeHeader.Flags**, the driver uses **InstanceIndex** as an index into the driver's list of static instance names for that block. WMI obtains the index from registration data provided by the driver when it registered the block.
- If **WNODE_FLAG_STATIC_INSTANCE_NAMES** is clear in **WnodeHeader.Flags**, the driver uses the offset at **OffsetInstanceName** to locate the instance name string in the input **WNODE_SINGLE_ITEM**. **OffsetInstanceName** is the offset in bytes from the beginning of the structure to a USHORT which is the length of the instance name string in bytes (not characters). This length includes the NULL terminator if present, followed by the instance name string in Unicode.

If the driver cannot locate the specified instance, it must fail the IRP and return **STATUS_WMI_INSTANCE_NOT_FOUND**. In the case of an instance with a dynamic instance name, this status indicates that the driver does not "own" the instance. WMI can therefore continue to query other data providers and return an appropriate error to the data consumer if another provider finds the instance but cannot handle the request for some other reason.

If the driver locates the instance and can handle the request, it sets the data item in the instance to the value in the **WNODE_SINGLE_ITEM**. If the data item is read-only, the driver leaves the item unchanged, fails the IRP and returns **STATUS_WMI_READ_ONLY**.

If the instance is valid but the driver cannot handle the request, it can return any appropriate error status.

See Also

DpWmiSetDataItem, **IoWmiRegistrationControl**, **WMI_LIB_CONTEXT**, **WmiSystemControl**,

WNODE_SINGLE_ITEM

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

IRP_MN_DISABLE_COLLECTION

[This is preliminary documentation and subject to change.]

Any WMI driver that registers one or more of its data blocks as expensive to collect must handle this IRP.

When Sent

WMI sends this IRP to request the driver to stop accumulating data for a data block that the driver registered as expensive to collect and for which data collection has been enabled.

WMI sends this IRP at **IRQL PASSIVE_LEVEL** in an arbitrary thread context.

Input

Parameters.WmiProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters.WmiDataPath points to a GUID that identifies the data block for which data accumulation should be stopped.

Output

None.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to **STATUS_SUCCESS** or to an appropriate error status such as the following:

STATUS_WMI_GUID_NOT_FOUND

STATUS_INVALID_DEVICE_REQUEST

On success, a driver sets **Irp->IoStatus.Information** to zero.

Operation

A driver registers a data block as expensive to collect by setting **WMIREG_FLAG_EXPENSIVE** in the **Flags** member of the **WMIREGGUID** or **WMI GUIDREGINFO** structure that the driver passes to WMI when it registers or updates the data block. A driver need not accumulate data for such a block until it receives an explicit request to enable collection.

A driver that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its **WMLIB_CONTEXT** structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's **DpWmiFunctionControl** routine, or simply returns **STATUS_SUCCESS** to the caller if the driver does not define an entry point for such a routine.

A driver handles an **IRP_MN_DISABLE_COLLECTION** request only if **Parameters.Wmi.ProviderId** points to the same device object as the pointer that the driver passed to **IoWmiRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

Before handling the request, the driver determines whether **Parameters.Wmi.DataPath** points to a GUID that the driver supports. If not, the driver fails the IRP and returns **STATUS_WMI_GUID_NOT_FOUND**. If the data block is valid but was not registered with **WMIREG_FLAG_EXPENSIVE**, the driver can return **STATUS_SUCCESS** and take no further action.

It is unnecessary for the driver to check whether data collection is already disabled because WMI sends a single disable request for the data block when the last data consumer disables collection for that block. WMI will not send another disable request without an intervening request to enable.

See Also

DpWmiFunctionControl, **IoWmiRegistrationControl**, **IRP_MN_ENABLE_COLLECTION**, **WMLIB_CONTEXT**, **WMIREGGUID**, **WMI GUIDREGINFO**, **WmiSystemControl**

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

IRP_MN_DISABLE_EVENTS

[This is preliminary documentation and subject to change.]

Any WMI driver that registers one or more event blocks must handle this IRP.

When Sent

WMI sends this IRP to inform the driver that a data consumer has requested no further notification of an event.

WMI sends this IRP at **IRQL_PASSIVE_LEVEL** in an arbitrary thread context.

Input

Parameters.Wmi.ProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters.Wmi.DataPath points to a GUID that identifies the event block to disable.

Output

None.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irps->IoStatus.Status** and **Irps->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irps->IoStatus.Status** to **STATUS_SUCCESS** or to an appropriate error status such as the following:

STATUS_WMI_GUID_NOT_FOUND

STATUS_INVALID_DEVICE_REQUEST

On success, a driver sets **Irps->IoStatus.Information** to zero.

Operation

A driver that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its **WMLIB_CONTEXT** structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's **DpWmiFunctionControl** routine, or simply returns **STATUS_SUCCESS** to the caller if the driver does not define an entry point for such a routine.

A driver handles an **IRP_MN_DISABLE_EVENTS** request only if **Parameters.Wmi.ProviderId** points to the same device object as the pointer that the driver passed to **IoWmiRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

Before handling a request, the driver determines whether **Parameters.Wmi.DataPath** points to a GUID the driver supports. If not, the driver fails the IRP and returns **STATUS_WMI_GUID_NOT_FOUND**.

If the driver supports the event block, it disables the event for all instances of that block.

It is unnecessary for the driver to check whether events are already disabled for the event block because WMI sends a single disable request for that event block when the last data consumer disables the event. WMI will not send another disable request without an intervening request to enable.

For details about defining event blocks, see the *Kernel-Mode Drivers Design Guide*.

See Also

DpWmiFunctionControl, IoWmiRegistrationControl, IRP_MN_ENABLE_EVENTS,
WMI_LIB_CONTEXT, WmiSystemControl

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

IRP_MN_ENABLE_COLLECTION

[This is preliminary documentation and subject to change.]

Any WMI driver that registers one or more of its data blocks as expensive to collect must handle this IRP.

When Sent

WMI sends this IRP to request the driver to start accumulating data for a data block that the driver registered as expensive to collect.

WMI sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

Parameters. Wmi.ProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters. Wmi.DataPath points to a GUID that identifies the data block for which data is accumulated.

Output

None.

I/O Status Block

If the driver handles the IRP by calling WmiSystemControl, WMI sets Irpp->IoStatus.Status and Irpp->IoStatus.Information in the I/O status block.

Otherwise, the driver sets Irpp->IoStatus.Status to STATUS_SUCCESS or to an appropriate error status such as the following:

STATUS_WMI_GUID_NOT_FOUND

STATUS_INVALID_DEVICE_REQUEST

On success, a driver sets Irpp->IoStatus.Information to zero.

Operation

A driver registers a data block as expensive to collect by setting WMIREG_FLAG_EXPENSIVE in the Flags member of the WMIREGGUID or WMIGUIDREGINFO structure. The driver passes these structures to WMI when it registers or updates the data block. A driver need not accumulate data for such a block until it receives an explicit request to start data collection.

A driver that handles WMI IRPs by calling WMI library support routines calls WmiSystemControl with a pointer to its WMI_LIB_CONTEXT structure, a pointer to its device object, and a pointer to the IRP. WmiSystemControl calls the driver's DpWmiFunctionControl routine, or simply returns STATUS_SUCCESS to the caller if the driver does not define an entry point for such a routine.

A driver handles an IRP_MN_ENABLE_COLLECTION request only if **Parameters.** Wmi.ProviderId points to the same device object as the pointer that the driver passed to IoWmiRegistrationControl. Otherwise, the driver forwards the request to the next-lower driver.

Before handling a request, the driver should make sure that **Parameters.** Wmi.DataPath points to a GUID that the driver supports. If it does not, the driver should fail the IRP and return STATUS_WMI_GUID_NOT_FOUND. If the data block is valid but was not registered with WMIREG_FLAG_EXPENSIVE, the driver can return STATUS_SUCCESS and take no further action.

If the block is valid and was registered with WMIREG_FLAG_EXPENSIVE, the driver enables data collection for all instances of that data block.

It is unnecessary for the driver to check whether data collection is already enabled for the data block. WMI sends only a single request to enable a data block after the first data consumer enables the block. WMI will not send another request to enable without an intervening disable request.

See Also

DpWmiFunctionControl, IoWmiRegistrationControl, IRP_MN_DISABLE_COLLECTION,
WMI_LIB_CONTEXT, WMIREGGUID, WmiSystemControl

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

IRP_MN_ENABLE_EVENTS

[This is preliminary documentation and subject to change.]

Any WMI driver that registers one or more event blocks must handle this IRP.

When Sent

WMI sends this IRP to inform the driver that a data consumer has requested notification of an event.

WMI sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

Input

Parameters. **Wmi.ProviderId** points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters. **Wmi.DataPath** points to a GUID that identifies the event block to enable.

Parameters. **Wmi.BufferSize** indicates the size of the nonpaged buffer at **Parameters.** **Wmi.Buffer**, which must be greater than or equal to the size of **WNODE_HEADER**. A driver that does not register trace blocks (**WMIREG_FLAG_TRACED_GUID**) can ignore this parameter.

Parameters. **Wmi.Buffer** points to a **WNODE_HEADER** that indicates whether the event should be traced (**WMI_FLAGS_TRACED_GUID**) and provides a handle to the system logger. A driver that does not register trace blocks (**WMIREG_FLAG_TRACED_GUID**) can ignore this parameter.

Output

None.

I/O Status Block

If the driver handles the IRP by calling **Wmi.SystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to **STATUS_SUCCESS** or to an appropriate error status such as the following:

STATUS_WMI_GUID_NOT_FOUND

STATUS_INVALID_DEVICE_REQUEST

On success, a driver sets **Irp->IoStatus.Information** to zero.

Operation

A driver that handles WMI IRPs by calling WMI library support routines calls **Wmi.SystemControl** with a pointer to its **WMI_LIB_CONTEXT** structure, a pointer to its device object, and a pointer to the IRP. **Wmi.SystemControl** calls the driver's **DpWmiFunctionControl** routine, or simply returns **STATUS_SUCCESS** to the caller if the driver does not define an entry point for such a routine.

A driver handles an **IRP_MN_ENABLE_EVENTS** request only if **Parameters.** **Wmi.ProviderId** points to the same device object as the pointer that the driver passed to **IoWmiRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

Before the driver handles the request, it should determine whether **Parameters.** **Wmi.DataPath**

points to a GUID that the driver supports. If not, the driver fails the IRP and returns **STATUS_WMI_GUID_NOT_FOUND**.

If the driver supports the event block, it enables the event for all instances of that data block.

It is unnecessary for the driver to check whether events are already enabled for the event block because WMI sends a single request to enable for the event block when the first data consumer enables the event. WMI will not send another request to enable without an intervening disable request.

A driver that registers trace blocks (**WMIREG_FLAG_TRACED_GUID**) must also determine whether to send the event to WMI or to the system logger for tracing. If tracing is requested, **Parameters.** **Wmi.Buffer** points to a **WNODE_HEADER** structure in which **Flags** is set with **WNODE_FLAG_TRACED_GUID** and **HistoricalContext** contains a handle to the logger.

For details about defining event blocks, sending events, and tracing, see the *Kernel-Mode Drivers Design Guide*.

See Also

DpWmiFunctionControl, **IoWmiRegistrationControl**, **IRP_MN_DISABLE_EVENTS**.

WMI_LIB_CONTEXT, **Wmi.SystemControl**, **WNODE_EVENT_ITEM**, **WNODE_HEADER**

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

IRP_MN_EXECUTE_METHOD

[This is preliminary documentation and subject to change.]

All drivers that support methods within data blocks must handle this IRP.

When Sent

WMI sends this IRP to execute a method associated with a data block.

WMI sends this IRP at IRQL PASSIVE_LEVEL in an arbitrary thread context.

WMI will send an **IRP_MN_QUERY_SINGLE_INSTANCE** prior to sending an **IRP_MN_EXECUTE_METHOD**. If a driver supports **IRP_MN_EXECUTE_METHOD** it must have a **IRP_MN_QUERY_SINGLE_INSTANCE** handler for the same data block whose method is being executed.

Input

Parameters. **Wmi.ProviderId** points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters. **WMI.DataPath** points to a GUID that identifies the data block associated with the method to execute.

Parameters. **WMI.BufferSize** indicates the size of the nonpaged buffer at **Parameters.WMI.Buffer** which must be $\geq \text{sizeof}(WNODE_METHOD_ITEM)$ plus the size of any output data for the method.

Parameters. **WMI.Buffer** points to a **WNODE_METHOD_ITEM** structure in which **MethodID** indicates the identifier of the method to execute and **DataBlockOffset** indicates the offset in bytes from the beginning of the structure to the first byte of input data, if any. **Parameters.WMI.BufferSizeDataBlock** indicates the size in bytes of the input **WNODE_METHOD_ITEM** including input data, or zero if there is no input.

Output

If the driver handles WMI IRPs by calling **WmiSystemControl**, WMI fills in the **WNODE_METHOD_ITEM** with data returned by the driver's **DpWmiExecuteMethod** routine.

Otherwise, the driver fills in the **WNODE_METHOD_ITEM** structure that **Parameters.WMI.Buffer** points to as follows:

- Updates **WnodeHeader.BufferSize** with the size of the output **WNODE_METHOD_ITEM**, including any output data.
- Updates **SizeDataBlock** with the size of the output data, or zero if there is no output data.
- Checks **Parameters.WMI.BufferSize** to determine whether the buffer is large enough to receive the output **WNODE_METHOD_ITEM** including any output data. If the buffer is not large enough, the driver fills in the needed size in a **WNODE_TOO_SMALL** structure pointed to by **Parameters.WMI.Buffer**. If the buffer is smaller than **sizeof(WNODE_TOO_SMALL)**, the driver fails the IRP and returns **STATUS_BUFFER_TOO_SMALL**.
- Writes output data, if any, over input data starting at **DataBlockOffset**. The driver must not change the input value of **DataBlockOffset**.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to **STATUS_SUCCESS** or to an appropriate error status such as the following:

STATUS_BUFFER_TOO_SMALL

STATUS_WMI_GUID_NOT_FOUND

STATUS_WMI_INSTANCE_NOT_FOUND

STATUS_WMI_ITEM_ID_NOT_FOUND

On success, a driver sets **Irp->IoStatus.Information** to the number of bytes written to the buffer at **Parameters.WMI.Buffer**.

Operation

A driver that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its **WMLIB_CONTEXT** structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's **DpWmiExecuteMethod** routine, or returns **STATUS_INVALID_DEVICE_REQUEST** to the caller if the driver does not define an entry point for such a routine.

A driver handles an IRP_MN_EXECUTE_METHOD request only if **Parameters.WMI.ProviderId** points to the same device object as the pointer that the driver passed to **IoWmiRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

Before handling the request, the driver determines whether **Parameters.WMI.DataPath** points to a GUID supported by the driver. If not, the driver fails the IRP and returns **STATUS_WMI_GUID_NOT_FOUND**.

If the driver supports the data block, it checks the input **WNODE_METHOD_ITEM** at **Parameters.WMI.Buffer** for the instance name, as follows:

- If **WNODE_FLAG_STATIC_INSTANCE_NAMES** is set in **WnodeHeader.Flags**, the driver uses **InstanceIndex** as an index into the driver's list of static instance names for that block. WMI obtains the index from registration data that was provided by the driver when it registered the block.
- If **WNODE_FLAG_STATIC_INSTANCE_NAMES** is clear in **WnodeHeader.Flags**, the driver uses the offset at **OffsetInstanceName** to locate the instance name string in the input **WNODE_METHOD_ITEM**. **OffsetInstanceName** is the offset in bytes from the beginning of the structure to a USHORT which is the length of the instance name string in bytes (not characters), including the NUL terminator if present, followed by the instance name string in Unicode.

If the driver cannot locate the specified instance, it must fail the IRP and return **STATUS_WMI_INSTANCE_NOT_FOUND**. In the case of a driver with a dynamic instance name, this status indicates that the driver does not "own" the instance. WMI can therefore continue to query other data providers and return an appropriate error to the data consumer if another provider finds the instance but cannot handle the request for some other reason.

The driver then checks the method ID in the input **WNODE_METHOD_ITEM** to determine whether it is a valid method for that data block. If not, the driver fails the IRP and returns **STATUS_WMI_ITEM_ID_NOT_FOUND**.

If the method generates output, the driver should check the size of the output buffer in **Parameters.WMI.BufferSize** before performing any operation that might have side effects or that should not be performed twice. For example, if a method returns the values of a group of counters and then resets the counters, the driver should check the buffer size (and fail the IRP if the buffer is too small) before resetting the counters. This ensures that WMI can safely resend the request with a

larger buffer.

If the instance and method ID are valid and the buffer is adequate in size, the driver executes the method. If **SizeDataBlock** in the input **WNODE_METHOD_ITEM** is non-zero, the driver uses the data starting at **DataBlockOffset** as input for the method.

If the method generates output, the driver writes the output data to the buffer starting at **DataBlockOffset** and sets **SizeDataBlock** in the output **WNODE_METHOD_ITEM** to the number of bytes of output data. If the method has no output data, the driver sets **SizeDataBlock** to zero. The driver must not change the input value of **DataBlockOffset**.

If the instance is valid but the driver cannot handle the request, it can return any appropriate error status.

See Also

[DpWmiExecuteMethod, IoWmiRegistrationControl, WMLIB_CONTEXT, WmiSystemControl, WNODE_METHOD_ITEM](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

IRP_MN_QUERY_ALL_DATA

[This is preliminary documentation and subject to change.]

All drivers that support WMI must handle this IRP.

When Sent

WMI sends this IRP to query for all instances of a given data block.

WMI sends this IRP at **IRQL PASSIVE_LEVEL** in an arbitrary thread context.

Input

Parameters, **WML.ProviderId** in the driver's I/O stack location in the IRP points to the device object of the driver that should respond to the request.

Parameters, **WML.DataPath** points to a GUID that identifies the data block.

Parameters, **WML.BufferSize** indicates the maximum size of the nonpaged buffer at **Parameters**, **WML.Buffer**, which receives output data from the request. The buffer size must be greater than or equal to **sizeof WNODE_ALL_DATA** plus the sizes of instance names and data for all instances to be returned.

Output

If the driver handles WMI IRPs by calling **WmiSystemControl**, WMI fills in a **WNODE_ALL_DATA** by calling the driver's **DpWmiQueryDataBlock** routine once for each block registered by the driver.

Otherwise, the driver fills in a **WNODE_ALL_DATA** structure at **Parameters**, **WML.Buffer** as follows:

- Sets **WnodeHeader.BufferSize** to the number of bytes of the entire **WNODE_ALL_DATA** to be returned, sets **WnodeHeader.Timestamp** to the value returned by **KeQuerySystemTime**, and sets **WnodeHeader.Flags** as appropriate for the data to be returned.
- Sets **InstanceCount** to the number of instances to be returned.
- If the block uses dynamic instance names, sets **OffsetInstanceNameOffsets** to the offset in bytes from the beginning of the **WNODE_ALL_DATA** to an array of offsets to dynamic instance names.
- If all instances are the same size:
 - Sets **WNODE_FLAG_FIXED_INSTANCE_SIZE** in **WnodeHeader.Flags** and sets **FixedInstanceSize** to that size, in bytes.
 - Writes instance data starting at **DataBlockOffset**, with padding so that each instance is aligned to an 8-byte boundary. For example, if **FixedInstanceSize** is 6, the driver adds 2 bytes of padding between instances.
- If instances vary in size:
 - Clears **WNODE_FLAG_FIXED_INSTANCE_SIZE** in **WnodeHeader.Flags** and writes an array of **InstanceCount** **OFFSETINSTANCEDATAANDLENGTH** structures starting at **OffsetInstanceDataAndLength**. Each **OFFSETINSTANCEDATAANDLENGTH** structure specifies the offset in bytes from the beginning of the **WNODE_ALL_DATA** structure to the beginning of the data for each instance, and the length of the data. **DataBlockOffset** is not used.
 - Writes instance data following the last element of the **OffsetInstanceDataAndLength** array, plus padding so that each instance is aligned to an 8-byte boundary.
- If the block uses dynamic instance names, writes the instance names at the offsets specified in the array at **OffsetInstanceNameOffsets**, with each dynamic name string aligned to a USHORT boundary.

If the buffer at **Parameters**, **WML.Buffer** is too small to receive all of the data, a driver fills in the needed size in a **WNODE_TOO_SMALL** structure at **Parameters**, **WML.Buffer**. If the buffer is smaller than **sizeof(WNODE_TOO_SMALL)**, the driver fails the IRP and returns **STATUS_BUFFER_TOO_SMALL**.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to **STATUS_SUCCESS** or to an appropriate error status such as the following:

STATUS_BUFFER_TOO_SMALL

STATUS_WMI_GUID_NOT_FOUND

On success, a driver sets **Irp->IoStatus.Information** to the number of bytes written to the buffer at **Parameters.WMI.Buffer**.

Operation

A driver that handles WMI IRPs by calling WMI library support routines calls **WmiSystemControl** with a pointer to its **WMI_LIB_CONTEXT** structure, a pointer to its device object, and a pointer to the **IRP**. **WmiSystemControl** calls the driver's **DpWmiQueryDataBlock** routine.

A driver handles an **IRP_MN_QUERY_ALL_DATA** request only if **Parameters.WMI.ProviderId** points to the same device object that the driver passed to **IoWmiRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

Before handling the request, the driver determines whether **Parameters.WMI.DataPath** points to a GUID that the driver supports. If not, the driver fails the IRP and returns **STATUS_WMI_GUID_NOT_FOUND**.

If the driver supports the data block, it fills in a **WNODE_ALL_DATA** structure at **Parameters.WMI.Buffer** with data for all instances of that data block.

See Also

[DpWmiQueryDataBlock](#), [IoWmiRegistrationControl](#), [KeQuerySystemTime](#), [WMI_LIB_CONTEXT](#), [WmiSystemControl](#), [WNODE_ALL_DATA](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

IRP_MN_QUERY_SINGLE_INSTANCE

[This is preliminary documentation and subject to change.]

All drivers that support WMI must handle this IRP.

When Sent

WMI sends this IRP to query for a single instance of a given data block.

WMI will send an **IRP_MN_QUERY_SINGLE_INSTANCE** prior to sending an **IRP_MN_EXECUTE_METHOD**. If a driver supports **IRP_MN_EXECUTE_METHOD** it must have a **IRP_MN_QUERY_SINGLE_INSTANCE** handler for the same data block whose method is being

executed.

WMI sends this IRP at **IRQL PASSIVE_LEVEL** in an arbitrary thread context.

Input

Parameters.WMI.ProviderId points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

Parameters.WMI.DataPath points to a GUID that identifies the data block to query.

Parameters.WMI.BufferSize indicates the maximum size of the nonpaged buffer at **Parameters.WMI.Buffer**, which points to a **WNODE_SINGLE_INSTANCE** structure that identifies the instance to query.

Output

If the driver handles WMI IRPs by calling **WmiSystemControl**, WMI fills in a **WNODE_SINGLE_INSTANCE** structure with data provided by the driver's **DpWmiQueryDataBlock** routine.

Otherwise, the driver fills in the **WNODE_SINGLE_INSTANCE** structure at **Parameters.WMI.Buffer** as follows:

- Updates **WnodeHeader.BufferSize** with the size in bytes of the output **WNODE_SINGLE_INSTANCE**, including instance data.
- Sets **SizeOfDataBlock** to the size in bytes of the instance data.
- Writes the instance data to **Parameters.WMI.Buffer** starting at **DataBlockOffset**. The driver must not change the input value of **DataBlockOffset**.

If the buffer at **Parameters.WMI.Buffer** is too small to receive all of the data, the driver fills in the needed size in a **WNODE_TOO_SMALL** structure at **Parameters.WMI.Buffer**. If the buffer is smaller than **sizeof(WNODE_TOO_SMALL)**, the driver fails the IRP and returns **STATUS_BUFFER_TOO_SMALL**.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, WMI sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to **STATUS_SUCCESS** or to an appropriate error status such as the following:

STATUS_BUFFER_TOO_SMALL

STATUS_WMI_GUID_NOT_FOUND

STATUS_WMI_INSTANCE_NOT_FOUND

On success, a driver sets `Irp->IoStatus.Information` to the number of bytes written to the buffer at `Parameters.WmiBuffer`.

Operation

A driver that handles WMI IRPs by calling WMI library support routines calls `WmiSystemControl` with a pointer to its `WMI_LIB_CONTEXT` structure, a pointer to its device object, and a pointer to the `IRP`. `WmiSystemControl` calls the driver's `DpWmiQueryDataBlock` routine.

A driver handles an `IRP_MN_QUERY_SINGLE_INSTANCE` request only if `Parameters.WmiProviderId` points to the same device object as the pointer that the driver passed in its call to `IoWmiRegistrationControl`. Otherwise, the driver forwards the request to the next-lower driver.

Before handling the request, the driver determines whether `Parameters.WmiDataPath` points to a GUID that the driver supports. If not, the driver fails the IRP and returns `STATUS_WMI_GUID_NOT_FOUND`.

If the driver supports the data block, it checks the input `WNODE_SINGLE_INSTANCE` at `Parameters.WmiBuffer` for the instance name, as follows:

- If `WNODE_FLAG_STATIC_INSTANCE_NAMES` is set in `WnodeHeader.Flags`, the driver uses `InstanceIndex` as an index into the driver's list of static instance names for that block. WMI obtains the index from registration data provided by the driver when it registered the block.
- If `WNODE_FLAG_STATIC_INSTANCE_NAMES` is clear in `WnodeHeader.Flags`, the driver uses the offset at `OffsetInstanceName` to locate the instance name string in the input `WNODE_SINGLE_INSTANCE`. `OffsetInstanceName` is the offset in bytes from the beginning of the structure to a USHORT which is the length of the instance name string in bytes (not characters), including the NULL terminator if present, followed by the instance name string in Unicode.

If the driver cannot locate the specified instance, it must fail the IRP and return `STATUS_WMI_INSTANCE_NOT_FOUND`. In the case of an instance with a dynamic instance name, this status indicates that the driver does not "own" the instance. WMI can therefore continue to query other data providers and return an appropriate error to the data consumer if another provider finds the instance but cannot handle the request for some other reason.

If the driver locates the instance and can handle the request, it fills in the `WNODE_SINGLE_INSTANCE` structure at `Parameters.WmiBuffer` with data for the instance.

If the instance is valid but the driver cannot handle the request, it can return any appropriate error status.

See Also

[DpWmiQueryDataBlock](#), [IoWmiRegistrationControl](#), [WMI_LIB_CONTEXT](#), [WmiSystemControl](#), [WNODE_SINGLE_INSTANCE](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

IRP_MN_REGINFO

[This is preliminary documentation and subject to change.]

All drivers that support WMI must handle this IRP.

When Sent

WMI sends this IRP to query or update a driver's registration information after the driver has called `IoWmiRegistrationControl`.

WMI sends this IRP at `IRQL_PASSIVE_LEVEL` in the context of a system thread.

Input

`Parameters.WmiProviderId` points to the device object of the driver that should respond to the request. This pointer is located in the driver's I/O stack location in the IRP.

`Parameters.WmiDataPath` is set to `WMIREGISTER` to query registration information or `WMIUPDATE` to update it.

`Parameters.WmiBufferSize` indicates the maximum size of the nonpaged buffer at `Parameters.WmiBuffer`. The size must be greater than or equal to the total of `(sizeof(WMIREGINFO) + (GuidCount * sizeof(WMIREGGUID)))`, where `GuidCount` is the number of data blocks and event blocks being registered by the driver, plus space for static instance names, if any.

Output

If the driver handles WMI IRPs by calling `WmiSystemControl`, WMI gets registration information for a driver's data blocks by calling its `DpWmiQueryRegInfo` routine.

Otherwise, the driver fills in a `WMIREGINFO` structure at `Parameters.WmiBuffer` as follows:

- Sets `BufferSize` to the size in bytes of the `WMIREGINFO` structure plus associated registration data.
- If the driver handles WMI requests on behalf of another driver, sets `NextWmiRegInfo` to the offset in bytes from the beginning of this `WMIREGINFO` to the beginning of another `WMIREGINFO` structure that contains registration information from the other driver.
- Sets `RegistryPath` to the registry path that was passed to the driver's `DriverEntry` routine.
- If `Parameters.WmiDataPath` is set to `WMIREGISTER`, sets `MofResourceName` to the offset from the beginning of this `WMIREGINFO` to a counted Unicode string that contains the name of the driver's MOF resource in its image file.

- Sets **GuidCount** to the number of data blocks and event blocks to register or update.
- Writes an array of **WMIREGGUID** structures, one for each data block or event block exposed by the driver, at **WmiRegGuid**.

The driver fills in each **WMIREGGUID** structure as follows:

- Sets **Guid** to the GUID that identifies the block.
- Sets **Flags** to provide information about instance names and other characteristics of the block. For example, if a block is being registered with static instance names, the driver sets **Flags** with the appropriate **WMIREG_FLAG_INSTANCE_XXX** flag.

If the block is being registered with static instance names, the driver:

- Sets **InstanceCount** to the number of instances.
- Sets one of the following members to an offset in bytes to static instance name data for the block:
 - If the driver sets **Flags** with **WMIREG_FLAG_INSTANCE_LIST**, it sets **InstanceNameList** to an offset to a list of static instance name strings. **WMI** specifies instances in subsequent requests by index into this list.
 - If the driver sets **Flags** with **WMIREG_FLAG_INSTANCE_BASENAME**, it sets **BaseNameOffset** to an offset to a base name string. **WMI** uses this string to generate static instance names for the block.
 - If the driver sets **Flags** with **WMIREG_FLAG_INSTANCE_PDO**, it sets **Pdo** to an offset to a pointer to the PDO passed to the driver's **AddDevice** routine. **WMI** uses the device instance path of the PDO to generate static instance names for the block.
- Writes the instance name strings, the base name string, or a pointer to the PDO at the offset indicated by **InstanceNameList**, **BaseName**, or **PDO**, respectively.

If the driver handles **WMI** registration on behalf of another driver (such as a miniclass or miniport driver), it fills in another **WMIREGINFO** structure with the other driver's registration information and writes it at **NextWmiRegInfo** in the previous structure.

If the buffer at **Parameters.WmiBuffer** is too small to receive all of the data, a driver writes the needed size in bytes as a **ULONG** to **Parameters.WmiBuffer** and fails the IRP and returns **STATUS_BUFFER_TOO_SMALL**.

I/O Status Block

If the driver handles the IRP by calling **WmiSystemControl**, **WMI** sets **Irp->IoStatus.Status** and **Irp->IoStatus.Information** in the I/O status block.

Otherwise, the driver sets **Irp->IoStatus.Status** to **STATUS_SUCCESS** or to an appropriate error status such as the following:

STATUS_BUFFER_TOO_SMALL

On success, a driver sets **Irp->IoStatus.Information** to the number of bytes written to the buffer at **Parameters.WmiBuffer**.

Operation

A driver that handles **WMI** IRPs by calling **WMI** library support routines calls **WmiSystemControl** with a pointer to its **WMI_LIB_CONTEXT** structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** calls the driver's **DpWmiQueryRegInfo** routine.

A driver handles an **IRP_MN_REGINFO** request only if **Parameters.Wmi.ProviderId** points to the same device object as the pointer that the driver passed to **IoWmiRegistrationControl**. Otherwise, the driver forwards the request to the next-lower driver.

Before handling the request, the driver checks **Parameters.Wmi.DataPath** to determine whether **WMI** is querying registration information (**WMIREGISTER**) or requesting an update (**WMIUPDATE**).

WMI sends this IRP with **WMIREGISTER** after a driver calls **IoWmiRegistrationControl** with **WMIREG_ACTION_REGISTER** or **WMIREG_ACTION_REREGISTER**. In response, a driver should fill in the buffer at **Parameters.Wmi.Buffer** with the following:

- A **WMIREGINFO** structure that indicates the driver's registry path, the name of its MOF resource, and the number of blocks to register.
- One **WMIREGGUID** structure for each block to register. If a block is to be registered with static instance names, the driver sets the appropriate **WMIREG_FLAG_INSTANCE_XXX** flag in the **WMIREGGUID** structure for that block.
- Any strings **WMI** needs to generate static instance names.

WMI sends this IRP with **WMIUPDATE** after a driver calls **IoWmiRegistrationControl** with **WMIREG_ACTION_UPDATE_GUID**. In response, a driver should fill in the buffer at **Parameters.Wmi.Buffer** with a **WMIREGINFO** structure as follows:

- To remove a block, the driver sets **WMIREG_FLAG_REMOVE_GUID** in its **WMIREGGUID** structure.
- To add or update a block (for example, to change its static instance names), the driver clears **WMIREG_FLAG_REMOVE_GUID** and provides new or updated registration values for the block.
- To register a new or existing block with static instance names, the driver sets the appropriate **WMIREG_FLAG_INSTANCE_XXX** and supplies any strings **WMI** needs to generate static instance names.

A driver can use the same **WMIREGINFO** structures to remove, add, or update blocks as it used initially to register all of its blocks, changing only the flags and data for the blocks to be updated. If a **WMIREGGUID** in such a **WMIREGINFO** structure matches exactly the **WMIREGGUID** passed by the driver when it first registered that block, **WMI** skips the processing involved in updating the block.

WMI does not send an **IRP_MN_REGINFO** request after a driver calls **IoWmiRegistrationControl** with **WMIREG_ACTION_DEREGISTER**, because **WMI** requires no further information from the driver. A driver typically deregisters its blocks in response to an **IRP_MN_REMOVE_DEVICE** request.

See Also

[DpWmiQueryRegInfo](#), [IoWmiRegistrationControl](#), [WMI_LIB_CONTEXT](#), [WMIREGGUID](#), [WMIREGINFO](#), [WmiSystemControl](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

Chapter 2 WMI Library Support Routines

[This is preliminary documentation and subject to change.]

This chapter describes the WMI library support routines that a driver can call to handle WMI IRPs. For information about handling WMI IRPs, see the *Kernel-mode Drivers Design Guide*.

For information about WMI library callback routines, see [Chapter 3](#).

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WmiCompleteRequest

[This is preliminary documentation and subject to change.]

```
NTSTATUS  
WmiCompleteRequest(  
    IN PDEVICE_OBJECT DeviceObject,  
    IN IRP Irp,  
    IN NTSTATUS Status,  
    IN ULONG BufferUsed,  
    IN CCHAR PriorityBoost  
);
```

WmiCompleteRequest indicates that a driver has finished processing a WMI request in a **DpWmiXxx** routine.

Parameters

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

Status

Specifies the status to return for the IRP.

BufferUsed

Specifies the number of bytes needed in the buffer passed to the driver's **DpWmiXxx** routine. If the buffer is too small, the driver sets **Status** to **STATUS_BUFFER_TOO_SMALL** and sets **BufferUsed** to the number of bytes needed for the data to be returned. If the buffer passed is large enough, the driver sets **BufferUsed** to the number of bytes actually used.

PriorityBoost

Specifies a system-defined constant by which to increment the runtime priority of the original thread that requested the operation. WMI calls **IoCompleteRequest** with **PriorityBoost** when it completes the IRP.

Return Value

WmiCompleteRequest returns the value that was passed to it in the **Status** parameter unless **Status** was set to **STATUS_BUFFER_TOO_SMALL**. If the driver set **Status** equal to **STATUS_BUFFER_TOO_SMALL**, **WmiCompleteRequest** builds a **WNODE_TOO_SMALL** structure and returns **STATUS_SUCCESS**. The return value from **WmiCompleteRequest** should be returned by the driver in its **DpWmiXxx** routine.

Comments

A driver calls **WmiCompleteRequest** from a **DpWmiXxx** routine after it finishes all other processing in that routine, or after the driver finishes all processing for a pending IRP. **WmiCompleteRequest** fills in a **WNODE_XXX** with any data returned by the driver and calls **IoCompleteRequest** to complete the IRP.

A driver should always return the return value from **WmiCompleteRequest** in its **DpWmiXxx** routine.

A driver must not call **WmiCompleteRequest** from its **DpWmiQueryRegInfo** routine.

Callers of **WmiCompleteRequest** must be running at **IRQL <= DISPATCH_LEVEL**.

See Also

[DpWmiExecuteMethod](#), [DpWmiFunctionControl](#), [DpWmiQueryDataBlock](#), [DpWmiQueryRegInfo](#), [DpWmiSetDataBlock](#), [DpWmiSetDataItem](#), [IoCompleteRequest](#), [WmiSystemControl](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WmiFireEvent

[This is preliminary documentation and subject to change.]

NTSTATUS

WmiFireEvent(


```

IN PDEVICE_OBJECT DeviceObject,
IN LPGUID Guid,
IN ULONG InstanceIndex,
IN ULONG EventDataSize,
IN PVOID EventData
);

```

WmiFireEvent sends an event to WMI for delivery to data consumers that have requested notification of the event.

Parameters

DeviceObject
Points to the driver's device object.

Guid
Points to the GUID that represents the event block.

InstanceIndex
If the event block has multiple instances, specifies the index of the instance.

EventDataSize
Specifies the number of bytes of data at *EventData*. If no data is generated for an event, *EventData* must be zero.

EventData
Points to a driver-allocated nonpaged buffer containing data generated by the driver for the event. If no data is generated for an event, *EventData* must be NULL. WMI frees the buffer without further intervention by the driver.

Return Value

WmiFireEvent propagates the status returned by **IoWmiWriteEvent**, or returns **STATUS_INSUFFICIENT_RESOURCES** if it could not allocate memory for the event.

Comments

A driver calls **WmiFireEvent** to send an event to WMI for delivery to all data consumers that have requested notification of the event. All parameters passed to **WmiFireEvent** must be allocated from nonpaged pool.

The driver sends an event only if it has been previously enabled by the driver's **DpWmiFunctionControl** routine, which WMI calls to process an **IRP_MN_ENABLE_EVENT** request.

The driver writes any data associated with the event to the buffer at *EventData*. WMI fills in a **WNODE_SINGLE_INSTANCE** structure with the data and calls **IoWmiWriteEvent** to deliver the event.

Callers of **WmiFireEvent** must be running at **IRQL <= DISPATCH_LEVEL**.

See Also

DpWmiFunctionControl, **IRP_MN_ENABLE_EVENTS**, **WmiSystemControl**

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WmiSystemControl

[This is preliminary documentation and subject to change.]

```

NTSTATUS
WmiSystemControl(
    IN PWMIIB_CONTEXT WmiLibInfo,
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    OUT PSYSCTL_IRP_DISPOSITION IrpDisposition
);

```

WmiSystemControl is a dispatch routine for drivers that use WMI library support routines to handle WMI IRPs.

Parameters

WmiLibInfo

Points to a **WMIIB_CONTEXT** structure that contains registration information for a driver's data blocks and event blocks and defines entry points for the driver's WMI library callback routines.

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

IrpDisposition

After **WmiSystemControl** returns, *IrpDisposition* indicates how the IRP was handled:

IrpProcessed

The IRP was processed and possibly completed. If the driver's **DpWmiXxx** routine called by **WmiSystemControl** did not complete the IRP, the driver must call **WmiCompleteRequest** to complete the IRP after **WmiSystemControl** returns.

IrpNotCompleted

The IRP was processed but not completed, either because WMI detected an error and set up the IRP with an appropriate error code, or processed an **IRP_MN_REGINFO** request. The driver must complete the IRP by calling **IoCompleteRequest**.

IrpNotWmi

The IRP is not a WMI request (that is, WMI does not recognize the IRP's minor code). If the driver handles **IRP_MJ_SYSTEM_CONTROL** requests with this **IRP_MN_XXX**, it should handle the IRP; otherwise the driver should forward the IRP to the next lower driver.

IrpForward

The IRP is targeted to another device object (that is, the device object pointer at **Parameters.Wmi.ProviderId** in the IRP does not match the pointer passed by the driver in its call to **IoWmiRegistrationControl**). The driver must forward the IRP to the next lower driver.

Return Value

WmiSystemControl returns **STATUS_SUCCESS** or one of the following error codes:

STATUS_INVALID_DEVICE_REQUEST

STATUS_WMI_GUID_NOT_FOUND

STATUS_WMI_INSTANCE_NOT_FOUND

Comments

When a driver receives an **IRP_MJ_SYSTEM_CONTROL** request with a WMI IRP minor code, it calls **WmiSystemControl** with a pointer to the driver's **WMLIB_CONTEXT** structure, a pointer to its device object, and a pointer to the IRP. The **WMLIB_CONTEXT** structure contains registration information for the driver's data blocks and event blocks and defines entry points for its WMI library callback routines.

WmiSystemControl confirms that the IRP is a WMI request and determines whether the block specified by the request is valid for the driver. If so, it processes the IRP by calling the appropriate **DpWmiXxx** entry point in the driver's **WMLIB_CONTEXT** structure. WMI is running at **IRQL_PASSIVE_LEVEL** when it calls the driver's **DpWmiXxx** routine.

Callers of **WmiSystemControl** must be running at **IRQL_PASSIVE_LEVEL**.

A driver must be running at **IRQL_PASSIVE_LEVEL** when it forwards an **IRP_MJ_SYSTEM_CONTROL** request to the next-lower driver.

See Also

[DpWmiExecuteMethod](#), [DpWmiFunctionControl](#), [DpWmiQueryDataBlock](#), [DpWmiQueryRegInfo](#), [DpWmiSetDataBlock](#), [DpWmiSetDataItem](#), [WMLIB_CONTEXT](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

Chapter 3 WMI Library Callback Routines

[This is preliminary documentation and subject to change.]

This describes required and optional routines that a driver must implement to handle WMI IRPs by calling WMI library support routines. A driver sets entry points to its **DpWmiXxx** routines in the

WMLIB_CONTEXT structure the driver passes to **WmiSystemControl**.

A driver's **DpWmiXxx** routines can have any names chosen by the driver writer.

For information about WMI library support routines, see [Chapter 2](#). For information about handling WMI IRPs, see the *Kernel-mode Drivers Design Guide*.

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

DpWmiExecuteMethod

[This is preliminary documentation and subject to change.]

NTSTATUS

DpWmiExecuteMethod(
IN **PDEVICE_OBJECT** *DeviceObject*,
IN **PIRP** *Irp*,
IN **ULONG** *GuidIndex*,
IN **ULONG** *InstanceIndex*,
IN **ULONG** *MethodId*,
IN **ULONG** *InBufferSize*,
IN **ULONG** *OutBufferSize*,
IN **OUT_PUCHAR** *Buffer*
);

A driver's **DpWmiExecuteMethod** routine executes a method associated with a data block. This routine is optional.

Parameters

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

GuidIndex

Specifies the data block by its index into the list of GUIDs provided by the driver in the **WMLIB_CONTEXT** structure it passed to **WmiSystemControl**.

InstanceIndex

If the block specified by *GuidIndex* has multiple instances, *InstanceIndex* specifies the instance.

MethodId

Specifies the ID of the method to execute. The driver defines the method ID as an item in a data block.

InBufferSize

Indicates the size in bytes of the input data. If there is no input data, *InBufferSize* is zero.

OutBufferSize

Buffer Indicates the number of bytes available in the buffer for output data.

Points to a buffer that holds the input data and receives the output data, if any, from the method. If the buffer is too small to receive all of the output, the driver returns `STATUS_BUFFER_TOO_SMALL` and calls `WmiCompleteRequest` with the size required.

Return Value

`DpWmiExecuteMethod` returns `STATUS_SUCCESS` or an appropriate error code such as the following:

`STATUS_BUFFER_TOO_SMALL`
`STATUS_INVALID_DEVICE_REQUEST`
`STATUS_WMI_INSTANCE_NOT_FOUND`
`STATUS_WMI_ITEM_ID_NOT_FOUND`

Comments

WMI calls a driver's `DpWmiExecuteMethod` routine after the driver calls `WmiSystemControl` in response to an `IRP_MN_EXECUTE_METHOD` request.

If a driver does not implement a `DpWmiExecuteMethod` routine, it must set `ExecuteWmiMethod` to `NULL` in the `WMI_LIB_CONTEXT` the driver passes to `WmiSystemControl`. In this case, WMI returns `STATUS_INVALID_DEVICE_REQUEST` to the caller in response to any `IRP_MN_EXECUTE_METHOD` request.

If the method generates output, the driver should check the size of the output buffer in `OutBufferSize` before performing any operation that might have side effects or that should not be performed twice. For example, if a method returns the values of a group of counters and then resets the counters, the driver should check the buffer size (and possibly return `STATUS_BUFFER_TOO_SMALL`) before resetting the counters. This ensures that WMI can safely re-send the request with a larger buffer.

After executing the method and writing output, if any, to the buffer, the driver calls `WmiCompleteRequest` to complete the request.

This routine can be pageable.

See Also

`IRP_MN_EXECUTE_METHOD`, `WMI_LIB_CONTEXT`, `WmiCompleteRequest`, `WmiSystemControl`

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

DpWmiFunctionControl

[This is preliminary documentation and subject to change.]

NTSTATUS

`DpWmiFunctionControl`(
IN `PDEVICE_OBJECT DeviceObject`,
IN `PIRP Irp`,
IN `ULONG GuidIndex`,
IN `WMIENABLEDISABLECONTROL Function`,
IN `BOOLEAN Enable`
);

A driver's `DpWmiFunctionControl` routine enables or disables notification of events. It also enables or disables data collection for data blocks that the driver registered as expensive to collect. This routine is optional.

Parameters

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

GuidIndex

Specifies the block by its index into the list of GUIDs provided by the driver in the `WMI_LIB_CONTEXT` structure it passed to `WmiSystemControl`.

Function

Specifies `WmiEventControl` to enable or disable an event, or `WmiDataBlockControl` to enable or disable data collection for a block that was registered as expensive to collect (that is, a block for which the driver set `WMIREG_FLAG_EXPENSIVE` in `Flags` of the `WMI_GUIDREGINFO` structure used to register the block).

Enable

Specifies `TRUE` to enable the event or data collection, or `FALSE` to disable it.

Return Value

`DpWmiFunctionControl` returns `STATUS_SUCCESS` or an appropriate error status such as:

`STATUS_WMI_GUID_NOT_FOUND`

`STATUS_INVALID_DEVICE_REQUEST`

Comments

WMI calls a driver's `DpWmiFunctionControl` routine after the driver calls `WmiSystemControl` in response to one of the following requests:

IRP_MN_ENABLE_COLLECTION
IRP_MN_DISABLE_COLLECTION
IRP_MN_ENABLE_EVENTS
IRP_MN_DISABLE_EVENTS

If a driver does not implement a `DpWmiFunctionControl` routine, it must set `WmiFunctionControl` to `NULL` in the `WMLIB_CONTEXT` the driver passes to `WmiSystemControl`. WMI returns `STATUS_SUCCESS` to the caller.

It is unnecessary for the driver to check whether events or data collection are already enabled for a block because WMI sends a single enable request when the first data consumer enables the block, and sends a single disable request when the last data consumer disables the block. WMI will not call `DpWmiFunctionControl` to enable a block without an intervening call to disable it.

After enabling or disabling the event or data collection for the block, the driver calls `WmiCompleteRequest` to complete the request.

This routine can be pageable.

See Also

IRP_MN_ENABLE_COLLECTION, IRP_MN_DISABLE_COLLECTION,
IRP_MN_ENABLE_EVENTS, IRP_MN_DISABLE_EVENTS, WMLIB_CONTEXT,
WmiSystemControl

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

DpWmiQueryDataBlock

[This is preliminary documentation and subject to change.]

NTSTATUS

`DpWmiQueryDataBlock(
IN PDEVICE_OBJECT DeviceObject,
IN PIRP Irp,
IN ULONG GuidIndex,
IN ULONG InstanceIndex,
IN ULONG InstanceCount,
IN OUT PULONG InstanceLengthArray,
IN OUT PULONG BufferAvail,
OUT PUCHAR Buffer`

);

A driver's `DpWmiQueryDataBlock` routine returns either a single instance or all instances of a data block. This routine is required.

Parameters

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

GuidIndex

Specifies the data block by its index into the list of GUIDs provided by the driver in the `WMLIB_CONTEXT` structure it passed to `WmiSystemControl`.

InstanceIndex

If `DpWmiQueryDataBlock` is called in response to an `IRP_MN_QUERY_SINGLE_INSTANCE` request, *InstanceIndex* specifies the instance to be queried. If `DpWmiQueryDataBlock` is called in response to an `IRP_MN_QUERY_ALL_DATA` request, *InstanceIndex* is zero.

InstanceCount

If `DpWmiQueryDataBlock` is called in response to an `IRP_MN_QUERY_SINGLE_INSTANCE` request, *InstanceCount* is 1. If `DpWmiQueryDataBlock` is called in response to an `IRP_MN_QUERY_ALL_DATA` request, *InstanceCount* is the number of instances to be returned.

InstanceLengthArray

Points to an array of ULONGs that indicate the length of each instance to be returned. If the buffer at *Buffer* is too small to receive all of the data, the driver sets *InstanceLengthArray* to `NULL`.

BufferAvail

Specifies the maximum number of bytes available to receive data in the buffer at *Buffer*.

Buffer

Points to the buffer to receive instance data. If the buffer is large enough to receive all of the data, the driver writes instance data to the buffer with each instance aligned on an 8-byte boundary. If the buffer is too small to receive all of the data, the driver calls `WmiCompleteRequest` with *BufferUsed* set to the size required.

Return Value

`DpWmiQueryDataBlock` returns `STATUS_SUCCESS` or an error status such as the following:

`STATUS_BUFFER_TOO_SMALL`

`STATUS_WMI_GUID_NOT_FOUND`

`STATUS_WMI_INSTANCE_NOT_FOUND`

If the driver cannot complete the request immediately, it can return `STATUS_PENDING`.

Comments

WMI calls a driver's `DpWmiQueryDataBlock` routine after the driver calls `WmiSystemControl` in response to an `IRP_MN_QUERY_DATA_BLOCK` or `IRP_MN_QUERY_ALL_DATA` request.

After writing instance data to the buffer, the driver calls `WmiCompleteRequest` to complete the request.

This routine can be pageable.

See Also

`IRP_MN_QUERY_ALL_DATA`, `IRP_MN_QUERY_SINGLE_INSTANCE`, `WMIIRB_CONTEXT`, `WmiCompleteRequest`, `WmiSystemControl`

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

DpWmiQueryReginfo

[This is preliminary documentation and subject to change.]

NTSTATUS

```
DpWmiQueryReginfo(
    IN PDEVICE_OBJECT DeviceObject,
    OUT PULONG RegFlags,
    OUT PUNICODE_STRING InstanceName,
    OUT PUNICODE_STRING *RegistryPath,
    OUT PUNICODE_STRING MofResourceName,
    OUT PDEVICE_OBJECT *Pdo
);
```

A driver's `DpWmiQueryReginfo` routine provides information about the data blocks and event blocks to be registered by a driver. This routine is required.

Parameters

DeviceObject

Points to the driver's device object.

RegFlags

Indicates common characteristics of all blocks being registered. Any flag set in `RegFlags` is applied to all blocks. A driver can supplement `RegFlags` for a given block by setting `Flags` in the block's `WMIGUIDREGINFO` structure. For example, a driver might clear `WMIREG_FLAG_EXPENSIVE` in `RegFlags`, but set it in `Flags` to register a given block as expensive to collect.

The driver sets one of the following flags in `RegFlags`:

- `WMIREG_FLAG_INSTANCE_BASENAME`

Requests WMI to generate static instance names from a base name provided by the driver at the `InstanceName`. WMI generates instance names by appending a counter to the base name.

WMIREG_FLAG_INSTANCE_PDO

Requests WMI to generate static instance names from the device instance ID for the PDO. If the driver sets this flag, it must also set `Pdo` to the PDO passed to the driver's `AddDevice` routine. WMI generates instance names from the device instance path of the PDO. Using the device instance path as a base for static instance names is efficient because such names are guaranteed to be unique. WMI automatically supplies a "friendly" name for the instance as an item in a data block that can be queried by data consumers.

A driver might also set one or more of the following flags in `RegFlags`, but more typically would set them in `Flags` of a block's `WMIGUIDREGINFO` structure:

WMIREG_FLAG_EVENT_ONLY_GUID

The blocks can be enabled or disabled as events only, and cannot be queried or set. If this flag is clear, the blocks can also be queried or set.

WMIREG_FLAG_EXPENSIVE

Requests WMI to send an `IRP_MN_ENABLE_COLLECTION` request the first time a data consumer opens a data block and an `IRP_MN_DISABLE_COLLECTION` request when the last data consumer closes the data block. This is recommended if collecting such data affects performance, because a driver need not collect the data until a data consumer explicitly requests it by opening the block.

WMIREG_FLAG_REMOVE_GUID

Requests WMI to remove support for the blocks. This flag is valid only in response to a request to update registration information (`IRP_MN_REGINFO` with `DataPath` set to `WMIUPDATE`).

InstanceName

Points to a single counted Unicode string that serves as the base name for all instances of all blocks to be registered by the driver. WMI frees the string with `ExFreePool`. If `WMIREG_FLAG_INSTANCE_BASENAME` is clear, `InstanceName` is ignored.

RegistryPath

Points to a counted Unicode string that specifies the registry path passed to the driver's `DriverEntry` routine.

MofResourceName

Points to a single counted Unicode string that indicates the name of the MOF resource attached to the driver's binary image file. This string can be a static string or one that the driver allocates. If the driver allocates the string, the driver is responsible for freeing the string. If the driver does not have a MOF resource attached, it can leave `MofResourceName` unchanged.

Pdo

Points to the physical device object (PDO) passed to the driver's `AddDevice` routine. If `WMIREG_FLAG_INSTANCE_PDO` is set, WMI uses the device instance path of this PDO as a base from which to generate static instance names. If `WMIREG_FLAG_INSTANCE_PDO` is clear, WMI ignores `Pdo`.

Return Value

`DpWmiQueryReginfo` always returns `STATUS_SUCCESS`

Comments

WMI calls a driver's `DpWmiQueryRegInfo` after the driver calls `WmiSystemControl` in response to an `IRP_MN_REGINFO` request. WMI sends this IRP after a driver calls `IoWmiRegistrationControl` with `WMIREG_ACTION_REGISTER`, `WMIREG_ACTION_REREGISTER`, or `WMIREG_ACTION_UPDATE`.

WMI does not send an `IRP_MN_REGINFO` request after a driver calls `IoWmiRegistrationControl` with `WMIREG_ACTION_DEREGISTER`, because WMI requires no further information from the driver. A driver typically deregisters its blocks in response to an `IRP_MN_REMOVE_DEVICE` request.

The driver provides new or updated registration information about individual blocks, or indicates blocks to remove, in the `WMLIB_CONTEXT` structure it passes to `WmiSystemControl`. After the initial call, which establishes the driver's registry path and MOF resource name, a driver's `DpWmiQueryRegInfo` routine can change flags common to all of a driver's blocks, provide a different base name string used to generate instance names, or change the basis for instance names from a string to the device instance path of the PDO.

The driver must not return `STATUS_PENDING` or block the request. The driver must not complete the request by calling `WmiCompleteRequest` from its `DpWmiQueryRegInfo` routine or by calling `IoCompleteRequest` after `WmiSystemControl` returns.

This routine can be pageable.

See Also

[IoWmiRegistrationControl](#), [IRP_MN_REGINFO](#), [WMLIB_CONTEXT](#), [WMI_GUID_REGINFO](#), [WmiSystemControl](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

DpWmiSetDataBlock

[This is preliminary documentation and subject to change.]

NTSTATUS

```
DpWmiSetDataBlock(  
    IN PDEVICE_OBJECT DeviceObject,  
    IN PIRP Irp,  
    IN ULONG GuidIndex,  
    IN ULONG InstanceIndex,  
    IN ULONG BufferSize,  
    IN PCHAR Buffer  
);
```

A driver's `DpWmiSetDataBlock` routine changes all data items in a single instance of a data block. This routine is optional.

Parameters

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

GuidIndex

Specifies the data block by its index into the list of GUIDs provided by the driver in the `WMLIB_CONTEXT` structure it passed to `WmiSystemControl`.

InstanceIndex

If the block specified by *GuidIndex* has multiple instances, *InstanceIndex* specifies the instance.

BufferSize

Specifies the size in bytes of the buffer at *Buffer*.

Buffer

Points to a buffer that contains new values for the instance.

Return Value

`DpWmiSetDataBlock` returns `STATUS_SUCCESS` or an appropriate error status such as the following:

`STATUS_WMI_INSTANCE_NOT_FOUND`

`STATUS_WMI_GUID_NOT_FOUND`

`STATUS_WMI_READ_ONLY`

`STATUS_WMI_SET_FAILURE`

If the driver cannot complete the request immediately, it can return `STATUS_PENDING`.

Comments

WMI calls a driver's `DpWmiSetDataItem` routine after the driver calls `WmiSystemControl` in response to an `IRP_MN_CHANGE_SINGLE_INSTANCE` request.

If a driver does not implement a `DpWmiSetDataItem` routine, it must set `SetWmiDataBlock` to `NULL` in the `WMLIB_CONTEXT` the driver passes to `WmiSystemControl`. WMI returns `STATUS_READ_ONLY` to the caller.

This routine can be pageable.

See Also

IRP_MN_CHANGE_SINGLE_INSTANCE, WMI_INSTANCE_CONTEXT, WMI_CONTEXT, WmiSystemControl

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

DpWmiSetDataItem

[This is preliminary documentation and subject to change.]

NTSTATUS

```
DpWmiSetDataItem(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp,
    IN ULONG GuidIndex,
    IN ULONG InstanceIndex,
    IN ULONG DataItemId,
    IN ULONG BufferSize,
    IN PCHAR Buffer
);
```

A driver's DpWmiSetDataItem changes a single data item in an instance of a data block. This routine is optional.

Parameters

DeviceObject

Points to the driver's device object.

Irp

Points to the IRP.

GuidIndex

Specifies the data block by its index into the list of GUIDs provided by the driver in the WMI_INSTANCE_CONTEXT structure it passed to WmiSystemControl.

InstanceIndex

If the block specified by *GuidIndex* has multiple instances, *InstanceIndex* specifies the instance.

DataItemId

Specifies the ID of the data item to set.

BufferSize

Specifies the size in bytes of the buffer at *Buffer*.

Buffer

Points to a buffer that contains the new value for the data item.

Return Value

DpWmiSetDataItem returns STATUS_SUCCESS or an appropriate error code such as the following:

STATUS_WMI_INSTANCE_NOT_FOUND

STATUS_WMI_INSTANCE_ID_NOT_FOUND

STATUS_WMI_GUID_NOT_FOUND

STATUS_WMI_READ_ONLY

STATUS_WMI_SET_FAILURE

Comments

WMI calls a driver's DpWmiSetDataItem routine after the driver calls WmiSystemControl in response to an IRP_MN_CHANGE_SINGLE_ITEM request.

If a driver does not implement a DpWmiSetDataItem routine, it must set SetWmiDataItem to NULL in the WMI_INSTANCE_CONTEXT the driver passes to WmiSystemControl. WMI returns STATUS_READ_ONLY to the caller.

This routine can be pageable.

See Also

IRP_MN_CHANGE_SINGLE_ITEM, WMI_INSTANCE_CONTEXT, WmiSystemControl

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

Chapter 4 WMI Structures

[This is preliminary documentation and subject to change.]

This describes, in alphabetic order, the structures that are used to pass WMI information between WMI and drivers that are kernel-mode data providers.

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WMI_INSTANCE_CONTEXT

[This is preliminary documentation and subject to change.]

```
typedef struct _WMI_INSTANCE_CONTEXT {
    ULONG GuidCount;
```

```

PWMI_GUIDREGINFO GuidList;
PWMI_QUERY_REGINFO QueryWmiRegInfo;
PWMI_QUERY_DATABLOCK QueryWmiDataBlock;
PWMI_SET_DATAITEM SetWmiDataItem;
PWMI_EXECUTE_METHOD ExecuteWmiMethod;
PWMI_FUNCTION_CONTROL WmiFunctionControl;
) WMLIB_CONTEXT, *PWMLIB_CONTEXT;

```

A WMLIB_CONTEXT structure provides registration information for a driver's data blocks and event blocks and defines entry points for the driver's WMI library callback routines.

Members

GuidCount

Specifies the number of blocks registered by the driver.

GuidList

Points to an array of **GuidCount** WMIGUIDREGINFO structures that contain registration information for each block.

QueryWmiRegInfo

Points to the driver's DpWmiQueryRegInfo routine, which is a required entry point for drivers that call WMI library support routines.

QueryWmiDataBlock

Points to the driver's DpWmiQueryDataBlock routine, which is a required entry point for drivers that call WMI library support routines.

SetWmiDataBlock

Points to the driver's DpWmiSetDataBlock routine, which is an optional entry point for drivers that call WMI library support routines. If the driver does not implement this routine, it must set this member to NULL. In this case, WMI returns STATUS_WMI_READ_ONLY to the caller in response to any IRP_MN_CHANGE_SINGLE_INSTANCE request.

SetWmiDataItem

Points to the driver's DpWmiSetDataItem routine, which is an optional entry point for drivers that call WMI library support routines. If the driver does not implement this routine, it must set this member to NULL. In this case, WMI returns STATUS_WMI_READ_ONLY to the caller in response to any IRP_MN_CHANGE_SINGLE_ITEM request.

ExecuteWmiMethod

Points to the driver's DpWmiExecuteMethod routine, which is an optional entry point for drivers that call WMI library support routines. If the driver does not implement this routine, it must set this member to NULL. In this case, WMI returns STATUS_INVALID_DEVICE_REQUEST to the caller in response to any IRP_MN_EXECUTE_METHOD request.

WmiFunctionControl

Points to the driver's DpWmiFunctionControl routine, which is an optional entry point for drivers that call WMI library support routines. If the driver does not implement this routine, it must set this member to NULL. In this case, WMI returns STATUS_SUCCESS to the caller in response to any IRP_MN_ENABLE_XXX or IRP_MN_DISABLE_XXX request.

Comments

A driver that handles WMI IRPs by calling WMI library support routines stores an initialized WMLIB_CONTEXT structure (or a pointer to such a structure) in its device extension. A driver can

use the same WMLIB_CONTEXT structure for multiple device objects if each device object supplies the same set of data blocks.

When the driver receives an IRP_MJ_SYSTEM_CONTROL request, it calls **WmiSystemControl** with a pointer to its WMLIB_CONTEXT structure, a pointer to its device object, and a pointer to the IRP. **WmiSystemControl** determines whether the IRP contains a WMI request and, if so, handles the request by calling the driver's appropriate DpWmiXxx routine.

Memory for this structure can be allocated from paged pool.

See Also

[DpWmiExecuteMethod](#), [DpWmiFunctionControl](#), [DpWmiQueryRegInfo](#), [DpWmiQueryDataBlock](#), [DpWmiSetDataBlock](#), [DpWmiSetDataItem](#), [WMIGUIDREGINFO](#), [WmiSystemControl](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WMIGUIDREGINFO

[This is preliminary documentation and subject to change.]

```

typedef struct {
    LPCGUID Guid;
    ULONG InstanceCount;
    ULONG Flags;
} WMIGUIDREGINFO, *PWMI_GUIDREGINFO;

```

A WMIGUIDREGINFO structure contains registration information for a given data block or event block exposed by a driver that uses the WMI library support routines.

Members

Guid

Points to the GUID that identifies the block. The memory that contains the GUID can be paged unless it is also used to call **WmiFireEvent**.

InstanceCount

Specifies the number of instances defined for the block.

Flags

Indicates characteristics of the block. WMI ORs **Flags** with the flags set by the driver in the *RegFlags* parameter of its DpWmiQueryRegInfo routine, which apply to all of the data blocks and event blocks registered by the driver. **Flags** therefore supplements the driver's default settings for a given block.

A driver might set the following flag in **Flags**:

WMIREG_FLAG_INSTANCE_PDO

Requests WMI to generate static instance names from the device instance ID for the PDO. If this flag is set, the *Pdo* parameter of the driver's DpWmiQueryRegInfo routine

points to the PDO passed to the driver's AddDevice routine. WMI generates instance names from the device instance path of the PDO. Using the device instance path as a base for static instance names is efficient because such names are guaranteed to be unique. WMI automatically supplies a "friendly" name for the instance as an item in a data block that can be queried by data consumers.

A driver might also set one or more of the following flags:

WMIREG_FLAG_EVENT_ONLY_GUID

The block can be enabled or disabled as an event only, and cannot be queried or set. If this flag is clear, the block can also be queried or set.

WMIREG_FLAG_EXPENSIVE

Requests WMI to send an IRP_MN_ENABLE_COLLECTION request the first time a data consumer opens the data block and an IRP_MN_DISABLE_COLLECTION request when the last data consumer closes the data block. This is recommended if collecting such data affects performance, because a driver need not collect the data until a data consumer explicitly requests it by opening the block.

WMIREG_FLAG_REMOVE_GUID

Requests WMI to remove support for this block. This flag is valid only in response to a request to update registration information (IRP_MN_REGINFO with **DataPath** set to **WMIUPDATE**).

Comments

A driver that handles WMI IRPs by calling WMI library support routines builds an array of **WMIGUIDREGINFO** structures, one for each data block and event block to be registered. The driver sets the **GuidList** member of its **WMILIB_CONTEXT** structure to point to the first **WMIGUIDREGINFO** in the series.

Memory for this structure can be allocated from paged pool.

See Also

[DpWmiQueryRegInfo](#), [IRP_MN_DISABLE_COLLECTION](#), [IRP_MN_ENABLE_COLLECTION](#), [IRP_MN_REGINFO](#), [WmiFireEvent](#), [WMILIB_CONTEXT](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WMIREGGUID

[This is preliminary documentation and subject to change.]

```
typedef struct {
    GUID Guid;
    ULONG Flags;
    ULONG InstanceCount;
    union {
        ULONG InstanceNameList;
```

```
    ULONG BaseNameOffset;
    ULONG_PTR Pdo;
    ULONG_PTR InstanceInfo;
};
) WMIREGGUID, *PWMIREGGUID
```

A **WMIREGGUID** contains new or updated registration information for a data block or event block.

Members

Guid

Specifies the GUID that represents the block to register or update.

Flags

Indicates characteristics of the block to register or update.

If a block is being registered with static instance names, a driver sets one of the following flags:

WMIREG_FLAG_INSTANCE_LIST

Indicates that the driver provides static instance names for this block in a static list following the **WMIREGINFO** structure in the buffer at **IrpStack->Parameters.WMI.Buffer**. If this flag is set, **InstanceNameList** is the offset in bytes from the beginning of the **WMIREGINFO** structure that contains this **WMIREGGUID** to a contiguous series of **InstanceCount** counted UNICODE strings.

WMIREG_FLAG_INSTANCE_BASENAME

Requests WMI to generate static instance names from a base name provided by the driver following the **WMIREGINFO** structure in the buffer at **IrpStack->Parameters.WMI.Buffer**. WMI generates instance names by appending a counter to the base name. If this flag is set, **BaseNameOffset** is the offset in bytes from the beginning of the **WMIREGINFO** structure that contains this **WMIREGGUID** to a single counted UNICODE string that serves as the base name.

WMIREG_FLAG_INSTANCE_PDO

Requests WMI to generate static instance names from the device instance ID for the PDO. If this flag is set, **InstanceInfo** points to the PDO passed to the driver's AddDevice routine. WMI generates instance names from the device instance path of the PDO. Using the device instance path as a base for static instance names is efficient because such names are guaranteed to be unique. WMI automatically supplies a "friendly" name for the instance as an item in a data block that can be queried by data consumers.

If a block is being registered with dynamic instance names,

WMIREG_FLAG_INSTANCE_LIST, **WMIREG_FLAG_INSTANCE_BASENAME**, and **WMIREG_FLAG_INSTANCE_PDO** must be clear.

A driver might also set one or more of the following flags:

WMIREG_FLAG_EVENT_ONLY_GUID

The block can be enabled or disabled as an event only, and cannot be queried or set. If this flag is clear, the block can also be queried or set.

WMIREG_FLAG_EXPENSIVE

Requests WMI to send an **IRP_MN_ENABLE_COLLECTION** request the first time a data consumer opens the data block and an **IRP_MN_DISABLE_COLLECTION** request when the last data consumer closes the data block. This is recommended if collecting such data affects performance, because a driver need not collect the data until a data

consumer explicitly requests it by opening the block.

WMIREG_FLAG_REMOVE_GUID
Requests WMI to remove support for this block. This flag is valid only in response to a request to update registration information (IRP_MN_REGINFO with **DataPath** set to **WMIUPDATE**).

WMIREG_FLAG_TRACED_GUID

The block can be written only to a log file and can be accessed only through user-mode routines declared in *wintrace.h*. Only NT kernel-mode data providers set this flag.

WMIREG_FLAG_TRACE_CONTROL_GUID
The GUID acts as the control GUID for enabling or disabling the trace GUIDs associated with it in the MOF file. This flag is valid only if **WMIREG_FLAG_TRACED_GUID** is also set. Only NT kernel-mode data providers set this flag.

InstanceCount

Specifies the number of static instance names to be defined for this block. If the block is being registered with dynamic instance names, WMI ignores **InstanceCount**.

InstanceNameList

Indicates the offset in bytes from the beginning of the **WMIREGINFO** structure that contains this **WMIREGGUID** to a contiguous series of **InstanceCount** counted Unicode strings. This member is valid only if **WMIREG_FLAG_INSTANCE_LIST** is set in **Flags**. If the block is being registered with dynamic instance names, WMI ignores **InstanceNameList**.

BaseNameOffset

Indicates the offset in bytes from the beginning of the **WMIREGINFO** structure that contains this **WMIREGGUID** to a single counted UNICODE string that serves as a base for WMI to generate static instance names. This member is valid only if **WMIREG_FLAG_INSTANCE_BASENAME** is set in **Flags**. If the block is being registered with dynamic instance names, WMI ignores **BaseNameOffset**.

Pdo

Points to the physical device object (PDO) passed to the driver's **AddDevice** routine. WMI uses the device instance path of this PDO as a base from which to generate static instance names. This member is valid only if **WMIREG_FLAG_INSTANCE_PDO** is set in **Flags**. If the block is being registered with dynamic instance names, WMI ignores **Pdo**.

InstanceInfo

Reserved for use by WMI.

Comments

A driver builds one or more **WMIREGGUID** structures in response to an **IRP_MN_REGINFO** request to register or update its blocks. The driver passes an array of such structures at the **WmiRegGuid** member of a **WMIREGINFO** structure, which the driver writes to the buffer at **IrpStack->Parameters.WmiBuffer**.

A driver can register or update a block with either static or dynamic instance names. Static instance names provide best performance; however, dynamic instance names are preferred for data blocks if the number of instances or instance names change frequently. For more information about instance names, see the *Kernel-mode Drivers Design Guide*.

See Also

[IRP_MN_REGINFO](#), [WMIREGINFO](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WMIREGINFO

[This is preliminary documentation and subject to change.]

```
typedef struct {
    ULONG BufferSize;
    ULONG NextWmiRegInfo;
    ULONG RegistryPath;
    ULONG MofResourceName;
    ULONG GuidCount;
    WMIREGGUID WmiRegGuid[];
} WMIREGINFO, *PWMIREGINFO;
```

A **WMIREGINFO** structure contains information provided by a driver to register or update its data blocks and event blocks.

Members

BufferSize

Indicates the total size of the WMI registration data associated with this **WMIREGINFO** structure, calculated as follows: **(sizeof(WMIREGINFO) + (GuidCount * sizeof(WMIREGGUID) + additional data)**. Additional data might include items such as the MOF resource name, registry path, and static instance names for blocks.

NextWmiRegInfo

If a driver handles WMI requests on behalf of another driver, as a class driver might on behalf of a miniclass driver, **NextWmiRegInfo** indicates the offset in bytes from the beginning of this **WMIREGINFO** to the next **WMIREGINFO** structure that contains WMI registration information for the other driver. Otherwise, **NextWmiRegInfo** is zero.

RegistryPath

Indicates the offset in bytes from the beginning of this structure to a counted Unicode string that specifies the registry path passed to the driver's **DriverEntry** routine. The string must be aligned on a USHORT boundary. This member should be set only in response to a WMI registration request (IRP_MN_REGINFO with **DataPath** set to **WMIREGISTER**).

MofResourceName

Indicates the offset in bytes from the beginning of this structure to a counted Unicode string that specifies the name of the MOF resource in the driver's image file. The string must be aligned on a USHORT boundary. This member should be set only in response to a WMI registration request (IRP_MN_REGINFO with **DataPath** set to **WMIREGISTER**).

GuidCount

Indicates the number of **WMIREGGUID** structures in the array at **WmiRegGuid**.

WmiRegGuid

Is an array of **GuidCount** **WMIREGGUID** structures.

Comments

In response to a registration request (IRP_MN_REGINFO with **DataPath** set to WMIREGISTER), a driver builds at least one WMIREGINFO structure and writes the WMIREGINFO structure to the buffer at **IrpStack->Parameters.WMI.Buffer**. The WMIREGINFO structure contains an array of WMIREGGUID structures, one for each data block or event block exposed by the driver.

If the driver handles WMI requests on behalf of another driver, it builds another WMIREGINFO containing an array of WMIREGGUID structures for each block exposed by the other driver, sets the **NextWmiRegInfo** member of the first WMIREGINFO to an offset in bytes from the beginning of the first WMIREGINFO to the beginning of the next WMIREGINFO in the buffer, and writes both structures to the buffer. The driver indicates the total size of both WMIREGINFO structures and associated data when calls **IoCompleteRequest** to complete the IRP.

A driver can use the same WMIREGINFO structure(s) to remove or update blocks in response to an update request (IRP_MN_REGINFO with **DataPath** set to WMIUPDATE). If **WMIREG_FLAG_REMOVE_GUID** is set in the **Flags** member of a WMIREGGUID, WMI removes that block from the list of blocks previously registered by the driver. If **WMIREG_FLAG_REMOVE_GUID** is clear, WMI updates registration information for that block only if other WMIREGGUID members have changed—otherwise, WMI does not change to its registration information for that block.

See Also

[IoCompleteRequest](#), [IRP_MN_REGINFO](#), [WMIREGGUID](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WNODE_ALL_DATA

[This is preliminary documentation and subject to change.]

```
typedef struct tagWNODE_ALL_DATA {
    struct _WNODE_HEADER WnodeHeader;
    ULONG DataBlockOffset;
    ULONG InstanceCount;
    ULONG OffsetInstanceNameOffsets;
    union {
        ULONG FixedInstanceSize;
        OFFSETINSTANCEDATAANDLENGTH OffsetInstanceDataAndLength[];
    };
} WNODE_ALL_DATA, *PWNODE_ALL_DATA;
```

A **WNODE_ALL_DATA** structure contains data for all instances of a data block or event block.

Members

WnodeHeader

Is a **WNODE_HEADER** structure that contains information common to all **WNODE_XXX**

structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the **WNODE_XXX** data being passed or returned.

DataBlockOffset

Indicates the offset in bytes from the beginning of the **WNODE_ALL_DATA** structure to the beginning of data for the first instance.

InstanceCount

Indicates the number of instances whose data follows the fixed members of the **WNODE_ALL_DATA** in the buffer at **IrpStack->Parameters.WMI.Buffer**.

OffsetInstanceNameOffsets

Indicates the offset in bytes from the beginning of the **WNODE_ALL_DATA** to an array of offsets to dynamic instance names. Each instance name must be aligned on a USHORT boundary. If all instances to be returned have static instance names, WMI ignores **OffsetInstanceNameOffsets**.

FixedInstanceSize

Indicates the size of each instance to be returned if all such instances are the same size. This member is valid only if the driver sets **WNODE_FLAG_FIXED_INSTANCE_SIZE** in **WnodeHeader.Flags**.

OffsetInstanceDataAndLength

If instances to be returned vary in size, **OffsetInstanceDataAndLength** is an array of **InstanceCount** **OFFSETINSTANCEDATAANDLENGTH** structures that specify the offset in bytes from the beginning of the **WNODE_ALL_DATA** to the beginning of each instance and its length. **OFFSETINSTANCEDATAANDLENGTH** is defined as follows:

```
typedef struct {
    ULONG OffsetInstanceData;
    ULONG LengthInstanceData;
} OFFSETINSTANCEDATAANDLENGTH, *POFFSETINSTANCEDATAANDLENGTH
```

OffsetInstanceData

Indicates the offset in bytes from the beginning of the **WNODE_ALL_DATA** to the instance data.

LengthInstanceData

Indicates the length in bytes of the instance data.

Each instance must be aligned on a USHORT boundary. The **OffsetInstanceDataAndLength** member is valid only if the driver clears **WNODE_FLAG_FIXED_INSTANCE_SIZE** in **WnodeHeader.Flags**.

Comments

A driver fills in a **WNODE_ALL_DATA** structure in response to an **IRP_MN_QUERY_ALL_DATA** request. A driver might also generate a **WNODE_ALL_DATA** as an event.

After filling in the fixed members of the structure, a driver writes instance data and dynamic instance names (if any) at **DataBlockOffset** and **OffsetInstanceNameOffsets**, respectively, in the buffer at **IrpStack->Parameters.WMI.Buffer**. If **WNODE_FLAG_FIXED_INSTANCE_SIZE** is clear, the first offset follows the last element of the **OffsetInstanceDataAndLength** array, plus padding so the data begins on an 8-byte boundary.

Instance names must be USHORT aligned. Instance data must be QUADWORD aligned.

See Also

[IRP_MN_QUERY_ALL_DATA](#), [WNODE_EVENT_ITEM](#), [WNODE_HEADER](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WNODE_EVENT_ITEM

[This is preliminary documentation and subject to change.]

```
typedef struct tagWNODE_EVENT_ITEM {
    struct _WNODE_HEADER WnodeHeader;
    // Rest of WNODE data indicated by flags in WnodeHeader
} WNODE_EVENT_ITEM, *PWNODE_EVENT_ITEM;
```

A WNODE_EVENT_ITEM contains data generated by a driver for an event.

WnodeHeader

Is a WNODE_HEADER structure that contains information common to all WNODE_XXX structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the WNODE_XXX data being passed or returned.

Comments

A WNODE_EVENT_ITEM contains whatever data the driver determines is appropriate for an event, in a WNODE_XXX structure that is appropriate for that data.

A driver generates only events that it has previously enabled in response to an IRP_MN_ENABLE_EVENTS request. To generate an event, a driver calls [IoWMIWriteEvent](#) and passes a pointer to the WNODE_EVENT_ITEM. WMI queues the event for delivery to all data consumers registered for that event.

For best performance, events should be small in size. However, if the amount of data for an event exceeds the maximum size defined in the registry, a driver can pass a WNODE_EVENT_REFERENCE, which WMI uses to query for the related WNODE_EVENT_ITEM. For more information about defining and generating WMI events, see the *Kernel-mode Drivers Design Guide*.

See Also

[IoWMIWriteEvent](#), [IRP_MN_ENABLE_EVENTS](#), [WNODE_ALL_DATA](#), [WNODE_EVENT_REFERENCE](#), [WNODE_HEADER](#), [WNODE_SINGLE_INSTANCE](#), [WNODE_SINGLE_ITEM](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WNODE_EVENT_REFERENCE

[This is preliminary documentation and subject to change.]

```
typedef struct tagWNODE_EVENT_REFERENCE {
    struct _WNODE_HEADER WnodeHeader;
    GUID TargetGuid;
    ULONG TargetDataBlockSize;
    union
    {
        ULONG TargetInstanceIndex;
        WCHAR TargetInstanceName[];
    };
} WNODE_EVENT_REFERENCE, *PWNODE_EVENT_REFERENCE;
```

A WNODE_EVENT_REFERENCE contains information that WMI can use to query for an event that exceeds the event size limit set in the registry.

Members

WnodeHeader

Is a WNODE_HEADER structure that contains information common to all WNODE_XXX structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the WNODE_XXX data being passed or returned.

TargetGuid

Indicates the GUID that represents the event to query.

TargetDataBlockSize

Indicates the size of the event.

TargetInstanceIndex

Indicates the index into the driver's list of static instance names for the event. This member is valid only if the event block was registered with static instance names and WNODE_FLAGS_STATIC_INSTANCE_NAMES is set in [WnodeHeader.Flags](#).

TargetInstanceName

Indicates the dynamic instance name of the event as a counted Unicode string. This member is valid only if WNODE_FLAGS_STATIC_INSTANCE_NAMES is clear in [WnodeHeader.Flags](#) and the event block was registered with dynamic instance names.

Comments

If the amount of data for an event exceeds the maximum size set in the registry, a driver can generate a WNODE_EVENT_REFERENCE that specifies a WNODE_EVENT_ITEM that WMI can query to obtain the event. For more information about defining and generating WMI events, see the *Kernel-mode Drivers Design*.

See Also

[WNODE_EVENT_ITEM](#), [WNODE_HEADER](#)

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WNODE_HEADER

[This is preliminary documentation and subject to change.]

```
typedef struct _WNODE_HEADER {
    ULONG BufferSize;
    UINT_PTR ProviderId;
    union {
        ULONG64 HistoricalContext;
        struct {
            ULONG Version;
            ULONG Linkage;
        };
    };
    union {
        HANDLE KernelHandle;
        LARGE_INTEGER Timestamp;
    };
    GUID Guid;
    ULONG ClientContext;
    ULONG Flags;
} WNODE_HEADER, *PWNODE_HEADER;
```

A **WNODE_HEADER** is the first member of all other **WNODE_XXX** structures. It contains information common to all such structures.

Members**BufferSize**

Specifies the size in bytes of the nonpaged buffer to receive any **WNODE_XXX** data to be returned, including this **WNODE_HEADER**, additional members of a **WNODE_XXX** structure of the type indicated by **Flags**, and any WMI- or driver-determined data that accompanies that structure.

ProviderId

Reserved for WMI.

HistoricalContext

Reserved for WMI.

Version

Reserved for WMI.

Linkage

Reserved for WMI.

TimeStamp

Indicates the system time a driver collected the **WNODE_XXX** data, in units of 100

nanoseconds since 1/1/1601. A driver can call **KeQuerySystemTime** to obtain this value. If the block is to be written to a log file (**WNODE_FLAG_LOG_WNODE**), an NT driver might also set **WNODE_FLAG_USE_TIMESTAMP** in **Flags** to request the system logger to leave the value of **TimeStamp** unchanged.

KernelHandle

Reserved for WMI.

Guid

Indicates the GUID that represents the data block associated with the **WNODE_XXX** to be returned.

ClientContext

Reserved for WMI.

Flags

Indicates the type of **WNODE_XXX** structure that contains the **WNODE_HEADER**:

WNODE_FLAG_ALL_DATA

The rest of a **WNODE_ALL_DATA** structure follows the **WNODE_HEADER** in the buffer.

WMI sets this flag in the **WNODE_HEADER** it passes with an **IRP_MN_QUERY_ALL_DATA** request.

A driver sets this flag in the **WNODE_HEADER** of an event that consists of all instances of a data block. If the data block size is identical for all instances, a driver also sets **WNODE_FLAG_FIXED_INSTANCE_SIZE**.

WNODE_FLAG_EVENT_ITEM

A driver sets this flag to indicate that the **WNODE_XXX** structure was generated as an event. This flag is valid only if **WNODE_FLAG_ALL_DATA**, **WNODE_FLAG_SINGLE_INSTANCE**, or **WNODE_FLAG_SINGLE_ITEM** is also set.

WNODE_FLAG_EVENT_REFERENCE

The rest of a **WNODE_EVENT_REFERENCE** structure follows the **WNODE_HEADER** in the buffer.

A driver sets this flag when it generates an event that is larger than the maximum size specified in the registry for an event. WMI uses the information in the **WNODE_EVENT_REFERENCE** to request the event data and schedules such a request according to the value of **WNODE_FLAG_SEVERITY_MASK**.

WNODE_FLAG_METHOD_ITEM

The rest of a **WNODE_METHOD_ITEM** structure follows the **WNODE_HEADER** in the buffer.

WMI sets this flag in the **WNODE_HEADER** it passes with an

IRP_MN_EXECUTE_METHOD request.

WNODE_FLAG_SINGLE_INSTANCE

The rest of a **WNODE_SINGLE_INSTANCE** structure follows the **WNODE_HEADER** in the buffer.

WMI sets this flag in the **WNODE_HEADER** it passes with a request to query or change an instance.

A driver sets this flag in the **WNODE_HEADER** of an event that consists of a single instance of a data block.

WNODE_FLAG_SINGLE_ITEM
The rest of a **WNODE_SINGLE_ITEM** structure follows the **WNODE_HEADER** in the buffer.

WMI sets this flag in the **WNODE_HEADER** it passes with a request to change an item.

A driver sets this flag in the **WNODE_HEADER** of an event that consists of a single data item.

WNODE_FLAG_TOO_SMALL
The rest of a **WNODE_TOO_SMALL** structure follows the **WNODE_HEADER** in the buffer.

A driver sets this flag when it passes a **WNODE_TOO_SMALL**, indicating that the buffer is too small for all of the **WNODE_XXX** data to be returned.

In addition, **Flags** might be set with one or more of the following flags that provide additional information about the **WNODE_XXX**:

WNODE_FLAG_FIXED_INSTANCE_SIZE
All instances of a data block are the same size. This flag is valid only if **WNODE_FLAG_ALL_DATA** is also set.

WNODE_FLAG_INSTANCES_SAME

The number of instances and the dynamic instance names in a **WNODE_ALL_DATA** to be returned are identical to those returned from the previous **WNODE_ALL_DATA** query. This flag is valid only if **WNODE_FLAG_ALL_DATA** is also set. This flag is ignored for data blocks registered with static instance names.

For optimized performance, a driver should set this flag if it can track changes to the number or names of its data blocks. WMI can then skip the processing required to detect and update dynamic instance names.

WNODE_FLAG_STATIC_INSTANCE_NAMES

The **WNODE_XXX** data to be returned does not include instance names.

WMI sets this flag before requesting **WNODE_XXX** data for data blocks registered with static instance names. After receiving the returned **WNODE_XXX** from the driver, WMI fills in the static instance names specified at registration before passing the returned **WNODE_XXX** to a data consumer.

WNODE_FLAG_PDO_INSTANCE_NAMES

Static instance names are based on the device instance ID of the PDO for the device. A driver requests such names by setting **WMIREG_FLAG_INSTANCE_PDO** in the **WMIREGGUID** it uses to register the block.

WMI sets this flag before requesting **WNODE_XXX** data for data blocks registered with PDO-based instance names.

WNODE_FLAG_SEVERITY_MASK

The driver-determined severity level of the event associated with a returned **WNODE_EVENT_REFERENCE**, with 0x00 indicating the least severe and 0xff

indicating the most severe level.

WMI uses the value of this flag to prioritize its requests for the event data.

WNODE_FLAG_USE_TIMESTAMP
The system logger should not modify the value of **TimeStamp** set by the driver.

An NT driver might also set **Flags** to one or more of the following values for event blocks to be written to a system log file:

WNODE_FLAG_LOG_WNODE

An event block is to be sent to the system logger. The event header is a standard **WNODE_HEADER** structure. If the driver clears **WNODE_FLAG_TRACED_GUID**, the block will also be sent to WMI for delivery to any data consumers that have enabled the event. The driver must allocate the **WNODE_XXX** from pool memory. WMI frees the memory after delivering the event to data consumers.

WNODE_FLAG_TRACED_GUID

An event block is to be sent only to the system logger. It does not get sent to WMI data consumers. The event header is an **EVENT_TRACE_HEADER** structure, declared in *eventtrace.h*, instead of a **WNODE_HEADER**. The driver must allocate memory for the **WNODE_XXX** and free it after **IoWMIWriteEvent** returns. The driver can allocate such memory either from the stack or, to minimize the overhead of allocating and freeing the memory, from the driver's thread local storage if the driver creates and maintains its own thread pool.

WNODE_FLAG_USE_GUID_PTR

The **Guid** member points to a GUID in memory, rather than containing the GUID itself. The system logger dereferences the pointer before passing the data to the consumer. This flag is valid only if **WNODE_FLAG_LOG_WNODE** or **WNODE_FLAG_TRACED_GUID** are also set.

WNODE_FLAG_USE_MOF_PTR

Data that follows the fixed members of a **WNODE_XXX** structure consists of an array of **MOF_FIELD** structures, defined in *eventtrace.h*, that contain pointers to data and sizes rather than the data itself. The array can contain up to **MAX_MOF_FIELD** elements. The system logger dereferences the pointers before passing the data to the consumer. This flag is valid only for blocks registered with **WMIREG_FLAG_TRACED_GUID**.

Comments

In an **IRP_MN_CHANGE_XXX** or **IRP_MN_EXECUTE_METHOD** request, **BufferSize** in the **IRP** indicates the maximum size in bytes of the output buffer, while **BufferSize** in the input **WNODE_HEADER** for such a request indicates the size in bytes of the input data in the buffer.

See Also

IoWMIWriteEvent, **KeQuerySystemTime**, **WNODE_ALL_DATA**, **WNODE_EVENT_ITEM**, **WNODE_EVENT_REFERENCE**, **WNODE_METHOD_ITEM**, **WNODE_SINGLE_INSTANCE**, **WNODE_SINGLE_ITEM**, **WNODE_TOO_SMALL**

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WNODE_METHOD_ITEM

[This is preliminary documentation and subject to change.]

```
typedef struct tagWNODE_METHOD_ITEM {
    struct _WNODE_HEADER WnodeHeader;
    ULONG OffsetInstanceName;
    ULONG InstanceIndex;
    ULONG MethodId;
    ULONG DataBlockOffset;
    ULONG SizeDataBlock;
    UCHAR VariableData[];
} WNODE_METHOD_ITEM, *PNODE_METHOD_ITEM;
```

A **WNODE_METHOD_ITEM** indicates a method associated with an instance of a data block and contains any input data for the method.

Members

WnodeHeader

Is a **WNODE_HEADER** structure that contains information common to all **WNODE_XXX** structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the **WNODE_XXX** data being passed or returned.

OffsetInstanceName

Indicates the offset in bytes from the beginning of this structure to the dynamic instance name of this instance, aligned on a USHORT boundary. This member is valid only if **WNODE_FLAG_STATIC_INSTANCE_NAMES** is clear in **WnodeHeader.Flags**. If the data block was registered with static instance names, WMI ignores **OffsetInstanceName**.

InstanceIndex

Indicates the index of this instance into the driver's list of static instance names for this data block. This member is valid only if the data block was registered with static instance names and **WNODE_FLAG_STATIC_INSTANCE_NAMES** is set in **WnodeHeader.Flags**. If the data block was registered with dynamic instance names, WMI ignores **InstanceIndex**.

MethodId

Specifies the ID of the method to execute.

DataBlockOffset

Indicates the offset from the beginning of an input **WNODE_METHOD_ITEM** to input data for the method, or the offset from the beginning of an output **WNODE_METHOD_ITEM** to output data from the method.

SizeDataBlock

Indicates the size of the input data in an input **WNODE_METHOD_ITEM**, or zero if there is no input. In an output **WNODE_METHOD_ITEM**, **SizeDataBlock** indicates the size of the output data, or zero if there is no output.

VariableData

Contains additional data, including the dynamic instance name if any, and the input for or output from the method aligned on an 8-byte boundary.

Comments

WMI passes a **WNODE_METHOD_ITEM** with an **IRP_MN_EXECUTE_METHOD** request to specify a method to execute in an instance of a data block, plus any input data required by the method.

If a method generates output, a driver overwrites the input data with the output at **DataBlockOffset** in the buffer at **IrpStack->Parameters.WMI.Buffer**, and sets **SizeDataBlock** in the **WNODE_METHOD_ITEM** to specify the size of the output data.

See Also

WNODE_HEADER

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WNODE_SINGLE_INSTANCE

[This is preliminary documentation and subject to change.]

```
typedef struct tagWNODE_SINGLE_INSTANCE {
    struct _WNODE_HEADER WnodeHeader;
    ULONG OffsetInstanceName;
    ULONG InstanceIndex;
    ULONG DataBlockOffset;
    ULONG SizeDataBlock;
    UCHAR VariableData[];
} WNODE_SINGLE_INSTANCE, *PNODE_SINGLE_INSTANCE;
```

A **WNODE_SINGLE_INSTANCE** contains values for all data items in one instance of a data block.

Members

WnodeHeader

Is a **WNODE_HEADER** structure that contains information common to all **WNODE_XXX** structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the **WNODE_XXX** data being passed or returned.

OffsetInstanceName

Indicates the offset from the beginning of this structure to the dynamic instance name of this instance, aligned on a USHORT boundary. This member is valid only if **WNODE_FLAG_STATIC_INSTANCE_NAMES** is clear in **WnodeHeader.Flags**. If the data block was registered with static instance names, WMI ignores **OffsetInstanceName**.

InstanceIndex

Indicates the index of an instance registered with static instance names. This member is valid only if **WNODE_FLAG_STATIC_INSTANCE_NAMES** is set in **WnodeHeader.Flags**. If the data block was registered with dynamic instance names, WMI ignores **InstanceIndex**.

DataBlockOffset

Indicates the offset from the beginning of this structure to the beginning of the instance.

SizeDataBlock

Indicates the size of the data block for this instance.

VariableData

Contains additional data, including the dynamic instance name if any, padding so the instance begins on an 8-byte boundary, and the instance of the data block to be returned.

Comments

WMI passes a **WNODE_SINGLE_INSTANCE** with an **IRP_MN_CHANGE_SINGLE_INSTANCE** request to set read-write data items in an instance of a data block. A driver can ignore values passed for read-only data items in the instance.

A driver fills in a **WNODE_SINGLE_INSTANCE** in response to an **IRP_MN_QUERY_SINGLE_INSTANCE** request or to generate an event that consists of a single instance.

See Also

WNODE_EVENT_ITEM, **WNODE_HEADER**

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WNODE_SINGLE_ITEM

[This is preliminary documentation and subject to change.]

```
typedef struct tagWNODE_SINGLE_ITEM {
    struct _WNODE_HEADER WnodeHeader;
    ULONG OffsetInstanceName;
    ULONG InstanceIndex;
    ULONG ItemId;
    ULONG DataBlockOffset;
    ULONG SizedDataItem;
    UCHAR VariableData[];
} WNODE_SINGLE_ITEM, *PNODE_SINGLE_ITEM;
```

A **WNODE_SINGLE_ITEM** contains the value of a single data item in an instance of a data block.

Members

WnodeHeader

Is a **WNODE_HEADER** structure that contains information common to all **WNODE_XXX** structures, such as the buffer size, the GUID that represents a data block associated with a request, and flags that provide information about the **WNODE_XXX** data being passed or returned.

OffsetInstanceName

Indicates the offset from the beginning of this structure to the dynamic instance name, if any,

aligned on a USHORT boundary. This member is valid only if **WNODE_FLAG_STATIC_INSTANCE_NAMES** is clear in **WnodeHeader.Flags**. If the data block was registered with static instance names, WMI ignores **OffsetInstanceName**.

InstanceIndex

Indicates the index into the driver's list of static instance names of this instance. This member is valid only if the data block was registered with static instance names and **WNODE_FLAG_STATIC_INSTANCE_NAME** is set in **WnodeHeader.Flags**. If the data block was registered with dynamic instance names, WMI ignores **InstanceIndex**.

ItemId

Specifies the ID of the data item to set.

DataBlockOffset

Indicates the offset from the beginning of this structure to the new value for the data item.

SizeDataItem

Indicates the size of the data item.

VariableData

Contains additional data, including the dynamic instance name if any, padding so the data value begins on an 8-byte boundary, and the new value for the data item.

Comments

WMI passes a **WNODE_SINGLE_ITEM** with an **IRP_MN_CHANGE_SINGLE_ITEM** request to set the value of a data item in an instance of a data block.

A driver builds a **WNODE_SINGLE_ITEM** to generate an event that consists of a single data item.

See Also

WNODE_EVENT_ITEM, **WNODE_HEADER**

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

WNODE_TOO_SMALL

[This is preliminary documentation and subject to change.]

```
typedef struct tagWNODE_TOO_SMALL {
    struct _WNODE_HEADER WnodeHeader;
    ULONG SizeNeeded;
} WNODE_TOO_SMALL, *PNODE_TOO_SMALL;
```

A **WNODE_TOO_SMALL** indicates the size of the buffer needed to receive output from a request.

Members

WnodeHeader

Is a **WNODE_HEADER** structure that contains information common to all **WNODE_XXX** structures, such as the buffer size, the GUID that represents a data block associated with a

request, and flags that provide information about the WNODE_XXX data being passed or returned.

SizeNeeded

Specifies the size of the buffer needed to receive all of the WNODE_XXX data to be returned.

Comments

When the buffer for a WMI request is too small to receive all of the data to be returned, a driver fills in a WNODE_TOO_SMALL structure to indicate the required buffer size. WMI can then increase the buffer to the recommended size and issue the request again. A driver is responsible for managing any side effects caused by handling the same request more than once.

See AlsoWNODE_HEADER

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

Chapter 5 WMI Event Trace Structures

[This is preliminary documentation and subject to change.]

This section describes the structure that is used to send WMI events to the WMI event logger.

Built on Tuesday, June 15, 1999

Kernel-Mode Drivers: Preliminary Windows 2000 DDK

EVENT_TRACE_HEADER

[This is preliminary documentation and subject to change.]

```
typedef struct _EVENT_TRACE_HEADER {
    USHORT Size;
    UCHAR HeaderType;
    UCHAR MarkerFlags;
    union {
        ULONG Version;
        struct {
            UCHAR Type;
            UCHAR Level;
            USHORT Version;
        } Class;
    };
    ULONGLONG ThreadId;
    LARGE_INTEGER Timestamp;
    union {
        GUID Guid;
    };
};
```

```
ULONGLONG GuidPtr;
};
union {
    struct {
        ULONG ClientContext;
        ULONG Flags;
    };
    struct {
        ULONG KernelTime;
        ULONG UserTime;
    };
    ULONG64 ProcessorTime;
} EVENT_TRACE_HEADER; *PEVENT_TRACE_HEADER;
```

An EVENT_TRACE_HEADER structure is used to pass a WMI event to the WMI event logger. It is overlaid on the WNODE_HEADER portion of the WNODE_EVENT_ITEM passed to IoWMIProcessEvent. Information contained in the EVENT_TRACE_HEADER is written to the WMI log file.

Members**Size**

Specifies the size in bytes of this structure. This value should be set to - SIZEOF (EVENT_TRACE_HEADER) plus the size of any driver data appended to the end of this structure. (Note: The size of this member is smaller than the size of the Size member of the WNODE_HEADER structure on which this structure is overlaid.)

HeaderType

Reserved for internal use.

MarkerFlags

Reserved for internal use.

Version

Drivers can use this member to store version information. This information is not interpreted by the event logger.

Class**Type**

Trace event type. This can be one of the predefined EVENT_TRACE_TYPE_XXX values contained in evnttrace.h or can be a driver defined value. Callers are free to define private event types with values greater than the reserved values in evnttrace.h.

Level

Trace instrumentation level. A driver defined value meant to represent the degree of detail of the trace instrumentation. Drivers are free to give this value meaning. This value should be 0 by default. More information on how consumers can request different levels of trace information will be provided in a future version of the documentation. **Version**

Version of trace record. Version information that can be used by the driver to track different event formats.

ThreadId

Reserved for internal use.

TimeStamp

Indicates the time the driver event occurred. This time is indicated in units of 100 nanoseconds since 1/1/1601. If the WNODE_FLAG_USE_TIMESTAMP is set in Flags, the system logger will leave the value of TimeStamp unchanged. Otherwise, the system logger will set the value

of **TimeStamp** at the time it receives the event. A driver can call **KcQuerySystemTime** to set the value of **TimeStamp**.

Guid

Indicates the GUID that identifies the data block for the event.

GuidPtr

If the **WNODE_FLAG_USE_GUID_PTR** is set in **Flags**, **GuidPtr** points to the GUID that identifies the data block for the event.

ClientContext

Reserved for internal use.

Flags

Provides information about the contents of this structure. For information on

EVENT_TRACE_HEADER **Flags** values, see the **Flags** description in **WNODE_HEADER**.

KernelTime

Reserved for internal use.

UserTime

Reserved.

ProcessorTime

Reserved for internal use.

Comments

A driver which supports trace events will use this structure to report events to the WMI event logger. Trace events should not be reported until the driver receives a request to enable events and the control GUID is one the driver supports. The driver should initialize an **EVENT_TRACE_HEADER** structure, fill in any user defined event data at the end and pass a pointer to the **EVENT_TRACE_HEADER** to **IoWmiWriteEvent**. The driver should continue reporting trace events until it receives a request to disable the control GUID for the trace events.

See Also

WNODE_HEADER, **WNODE_EVENT_ITEM**, **IoWmiWriteEvent**

Built on Tuesday, June 15, 1999